Universidad de Alcalá
Escuela Politécnica Superior

GRADO EN INGENIERIA ELECTRONICA DE COMUNICACIONES

**Trabajo Fin de Grado**

# Depuración con Tracealyzer de sistemas embebidos basados en FreeRTOS sobre plataformas Cortex-M3



**Autor:** Nelson Ismael Rivero Meneses

**Tutor:** José Manuel Villadangos Carrizo

2021

# UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

**GRADO EN INGENIERIA ELECTRONICA DE COMUNICACIONES**

Trabajo Fin de Grado

"Depuración con Tracealyzer de sistemas embebidos basados en FreeRTOS sobre plataformas Cortex-M3"

**Autor:** Nelson Ismael Rivero Meneses

**Tutor:** José Manuel Villadangos Carrizo

**TRIBUNAL:**

**Presidente:** Ignacio Fernández Lorenzo

**Vocal 1º:** José Luis Martín Sánchez

**Vocal 2º:** José Manuel Villadangos Carrizo

**FECHA**:

# Agradecimientos

A todas las personas que he conocido a lo largo de mis estudios, a toda mi familia, y en especial a mis padres porque, pese a todas las dificultades que se han presentado en el camino, siempre me enseñaron a perseverar.

# Índice principal

# Índice de figuras

# Índice de tablas

# 1. Resumen

El objetivo del proyecto se centra en describir la herramienta Tracealyzer, que se emplea para depurar sistemas embebidos basados en RTOS. La aplicación desarrollada sobre la que se muestra esta plataforma es un pulsómetro basado en un sensor digital MAX30102 sobre la tarjeta LPC1768-Mini-DK2 de la familia NXP con un microcontrolador Cortex-M3 (LPC1768). La herramienta de desarrollo ha sido el entorno Keil uVision5 y como depurador y programador flash el ULINKpro del fabricante ARM.

**Palabras clave:** *FreeRTOS, LPC1768, ULINKpro, Tracealycer, Sistema Operativo (SO).*

# 2. Abstract

The objective of this project is to describe the Tracealyzer tool, which is used to debug embedded systems based on RTOS. The application developed on which this platform is shown is a heart rate monitor based on a MAX30102 digital sensor on the LPC1768-Mini-DK2 card of the NXP family with a Cortex-M3 microcontroller (LPC1768). The development tool has been the Keil uVision5 environment and as a debugger and flash programmer the ULINKpro from the manufacturer ARM.

**Key words:** *FreeRTOS, LPC1768, ULINKpro, Tracealycer, Operating System (OS).*

# 3. Resumen extendido

La herramienta ***Tracealyzer [1]*** de Percepio ha sido empleada para depurar un sistema embebido basado en ***FreeRTOS*** ejecutando una aplicación que mide el nivel de oxígeno en sangre y las pulsaciones por minuto. ***Tracealyzer*** soporta una amplia gama de sistemas operativos de tiempo real, entre los que se encuentran ***FreeRTOS,*** *ARM Keil RTX5****,*** *distribuciones Linux basadas en Yocto, SafeRTOS, Azyre RTOS ThreadX, Micrium uC/OS III, On Time RTOS-32, OpenVX – Synopsys EV6x Wind River VxWorks*.

El sistema embebido real se basa en un pulsómetro, basado en el sistema operativo ***FreeRTOS,*** sobre el microcontrolador LPC1768 de NXP de la familia de microcontroladores Cortex-M3 de ARM junto a la pantalla LCD de 2.8 pulgadas, el circuito integrado MAX30102, y un buzzer.

El integrado MAX30102 mide el nivel de oxígeno en la sangre y las pulsaciones por minuto, y se comunica con el microcontrolador LPC1768, encargado de gestionar la información y mostrar al usuario los datos en la pantalla LCD y también en el PC a través de una conexión con el puerto serie. Adicionalmente, emite un pitido cada vez que realiza una medida completa a través del buzzer.


Figura 1. Diagrama de bloques del sistema embebido

Para gestionar los recursos hardware de la tarjeta *LPC1768-Mini-DK2*, se ha instalado el sistema operativo ***FreeRTOS*** en la tarjeta.

Se ha empleado el programador y depurador ***ULINKpro*** de ARM para ver la trazabilidad del microcontrolador. En la parte final se explican y exponen los distintos recursos que ofrece la herramienta ***Tracealyzer*** para depurar sistemas embebidos y visualizar distintos eventos del sistema, uso de memoria y carga de la CPU. Al final del documento se encuentra la bibliografía y un anexo que contiene el código del programa principal.

Es muy importante entender algunos conceptos que definen a un sistema operativo y la depuración para poder entender los resultados obtenidos a la vez que se puedan interpretar fácilmente.

- Se define ***Sistema Operativo*** como el software principal o código que gestiona recursos hardware del sistema electrónico a desarrollar y que provee servicios a los programas de aplicación.
- El ***Kernel*** se entiende como el nivel más bajo que interactúa con el hardware del CPU.
- Se entiende por ***depuración*** a la acción de analizar los errores de un programa y eliminarlos. Un error típico sucede cuando el programa no puede continuar ejecutándose y se bloquea debido a que se intenta acceder a una zona de memoria protegida o no disponible.
- Los depuradores tienen la capacidad de ejecutar un programa instrucción a instrucción, parar el programa en una zona en concreto y mostrar el valor de variables o zonas de memoria. La mayoría de los procesadores poseen en su diseño de CPU módulos hardware para hacer la depuración más fácil. El *JTAG* es un conector hardware muy común en interfaces de depuración en la arquitectura de ARM.

El objetivo principal es analizar la herramienta de depuración ***Tracealyzer*** sobre el sistema embebido real, el pulsómetro, y destacar sus recursos más útiles en cuanto a depuración.

Otro de los objetivos principales es identificar los distintos sistemas operativos para microcontroladores y seleccionar el más adecuado. Se ha elegido ***FreeRTOS*** por diversos motivos que se explican en el apartado 4.1.

Por último, se desea seleccionar un entorno de programación adecuado para desarrollar el software del pulsómetro, objetivo que se detalla en el apartado 4.5.

Un pulsómetro es un dispositivo electrónico que es capaz de medir la frecuencia cardíaca y la saturación del oxígeno en la sangre. Para conseguirlo, el dispositivo emite radiación de luz roja e infrarroja y, posteriormente, detecta la intensidad que se refleja en la piel. En función de las diferentes intensidades absorbidas en la piel se pueden establecer las pulsaciones por minuto (p.p.m.) y el nivel de oxígeno en la sangre del usuario.


Figura 2. Pulsómetro comercial [1]

# 4. Herramientas de desarrollo

## 4.1. Tarjeta de desarrollo basada en el Cortex-M3 LPC1768

La tarjeta de desarrollo que se ha empleado es la LPC1768-Mini-DK2 [2] , basada en el microcontrolador LPC1768 del fabricante NXP (siguiente figura). Esta tarjeta posee varios periféricos y varios puertos de alimentación por lo que brinda un abanico muy grande de posibilidades a la hora de desarrollar todo tipo de proyectos.

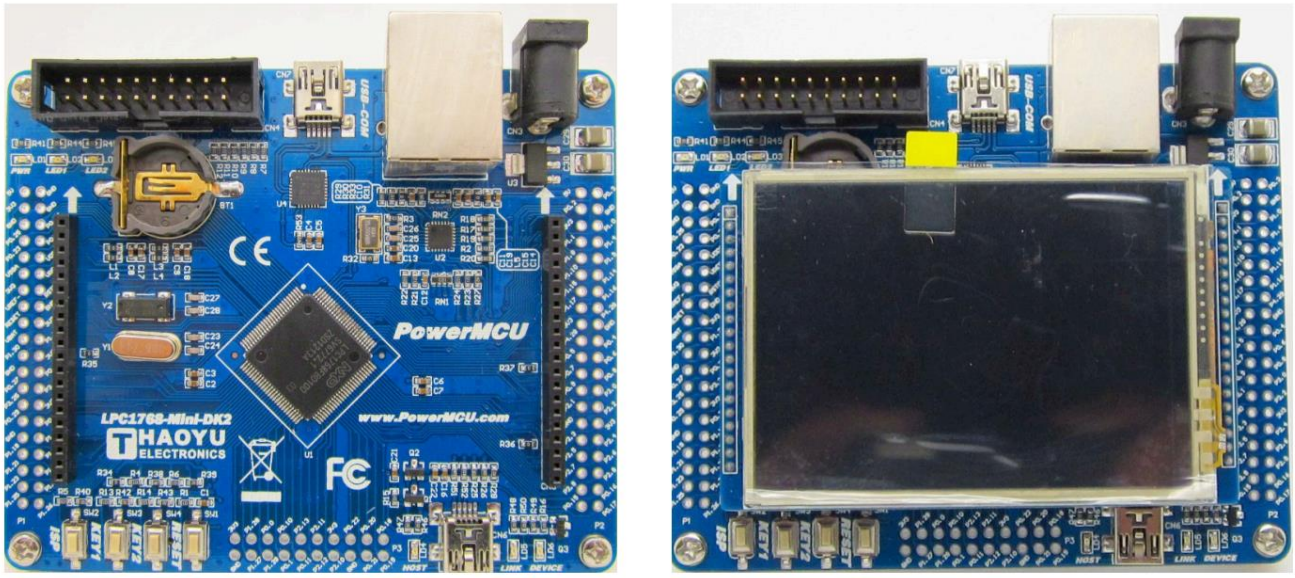



Figura 3. Tarjeta de desarrollo LPC1768-Mini-DK2 [2]

Las principales características de la tarjeta Mini-DK2 son:

- Microcontrolador Cortex-M3 LPC1768 de NXP
- 512KB de FLASH
- 64KB de SRAM
- Módulo de emulación de traza (ETM) que provee traza en tiempo real
- Interfaz estándar JTAG-SW (Joint Test Action Group)
- Conector serie para depuración (Serial Wire Debug) y conector serie para puerto específico para la trazabilidad (Serial Wire Trace Port)
- módulo Ethernet MAC, DMA, I2C x3, SSp/SPI x3, I2S, UART x4, RS485, BUS CAN x2, 12bit 8-ch ADC, 10-bit DAC, 4x32-bit Timers
- Control de motor por PWM, Lectura de Encoder por cuadratura

- posibilidad de trabajar como Modem
- Conexión COMx USB mediante conversor USB-serie integrado
- Jack para alimentación externa (5V)
- Pulsadores y LEDs
- Conector Paralelo y SPI para panel TFT a color táctil 2.8 pulgadas
- Tarjeta microSD
- USB Host/Device
- Adaptador Ethernet 100Mbit/s

## 4.2. Módulo pulsómetro basado en el MAX30102

El sistema sobre el que se realiza la depuración consiste en una pequeña aplicación que muestra al usuario por pantalla y por puerto serie el valor del nivel de oxígeno en la sangre y el número de pulsos por minuto de una persona.

Para conseguirlo se ha hecho uso del módulo pulsómetro basado en el circuito integrado ***MAX30102 [3]*** . Dicho integrado es capaz de medir el nivel de oxígeno en la sangre y las pulsaciones por minuto gracias a la tecnología fotónica, que consiste en la emisión, reflexión y detección de luz, nada invasivo para cualquier persona. El sensor capaz de trabajar en modo ultra-bajo consumo, ideal para dispositivos móviles que funcionen a pilas, dispositivos asistentes de fitness, smartphones o tablets. Puede operar en rango de temperaturas bastante amplio, desde -40°C hasta los 85°C. Las dimensiones del sensor son muy pequeñas, de 5.6mm x 3.3 mm x 1.55 mm en comparación con las dimensiones de la tarjeta donde se encuentra soldado, de 21 mm x 16 mm.

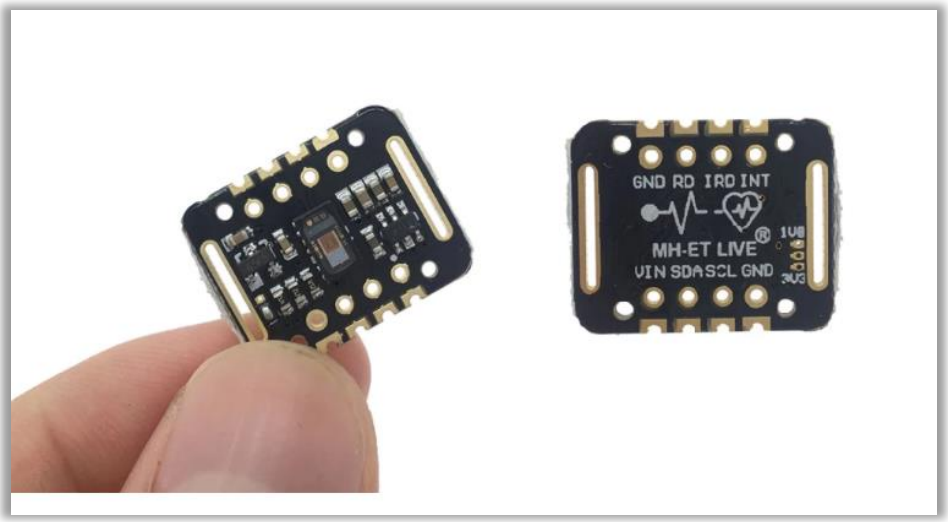



Figura 4. Módulo pulsómetro basado en integrado MAX30102 [3]

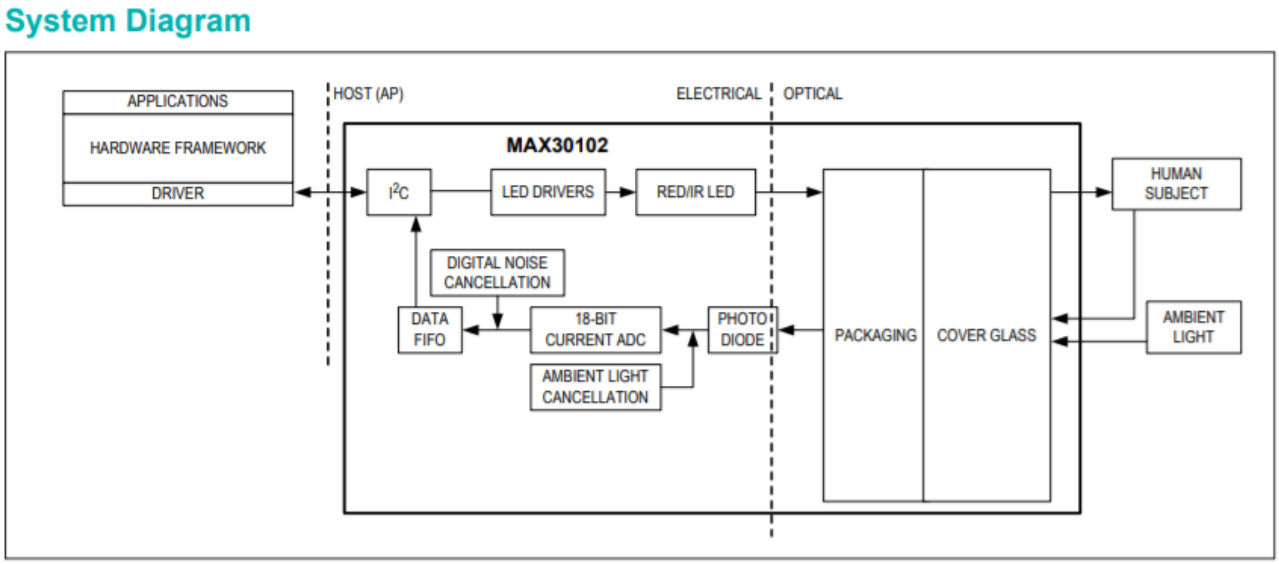



Figura 5. Diagrama del sensor [4]

El módulo posee la electrónica necesaria para conectar únicamente la alimentación del sistema y dos líneas para conectarse con el sensor vía ***i2c***. Adicionalmente, posee una línea de interrupción que puede emplearse opcionalmente.

El funcionamiento del sensor se basa en la detección de luz, por lo que es muy fácil de emplear en cualquier persona de cualquier edad sin causar la más mínima molestia. El modo de trabajo es muy simple, el integrado posee un diodo emisor de luz y un fotodetector. En primer lugar, es necesario activar el diodo emisor de luz para emitir radiación, dicha radiación se tiene que reflejar en la piel del paciente para que se pueda medir con el fotodetector.

El fotodetector se basa en una unión ***pn***, es decir, es un diodo que genera electrones en relación directa con los fotones que recibe, convirtiendo luz recibida en electricidad para que, posteriormente, se pueda cuantificar el nivel de electricidad empleando un **ADC** tipo ***sigma-delta***. Finalmente se obtiene el valor de la luz medida en forma de datos binarios que se guardan en una memoria ***FIFO***. El microcontrolador se comunica con el ***MAX30102*** para poder leer los valores medidos y realizar los cálculos necesarios de modo que se puede obtener el nivel de oxígeno que posee el usuario y los pulsos por minuto.

Esto es posible debido a que la luz incide en la piel buscando el nivel de hemoglobina que posee. La hemoglobina se encarga de transportar el oxígeno desde los órganos respiratorios hasta los tejidos del cuerpo humano. Dicha luz recibida se compara con la que se ha emitido de manera constante y se obtiene el nivel de oxígeno en sangre. Para medir el nivel de oxígeno en la sangre emplea un led rojo y otro infrarrojo, para medir los pulsos basta con el led rojo.

Por otro lado, la luz en general vibra a una velocidad y longitud de onda determinada, y la luz del led rojo e infrarrojo que emite el sensor vibra a 660 nm y 880nm respectivamente (ver figura 7). Dicha luz debe ser recibida, tras reflejarse en la piel, por el fotodiodo, que es capaz de detectar luz desde los 600 nm hasta los 900 nm sin ningún problema, según indica el fabricante en la hoja de características (ver figura 9).

La electrónica interna del sensor es capaz de realizar un filtrado que se muestra en la figura 9, cancelando el ruido del orden de los KHz, llamado cancelación de luz ambiente (**ALC**) de manera que permite el paso de luz a una frecuencia bastante baja, lo que tiene sentido porque el corazón normalmente late de 60 a 100 veces por minuto. Para que el sensor trabaje de manera óptima debe situarse a una distancia de 3 a 5 mm de la piel del usuario tal y como explica el fabricante en la figura 8.

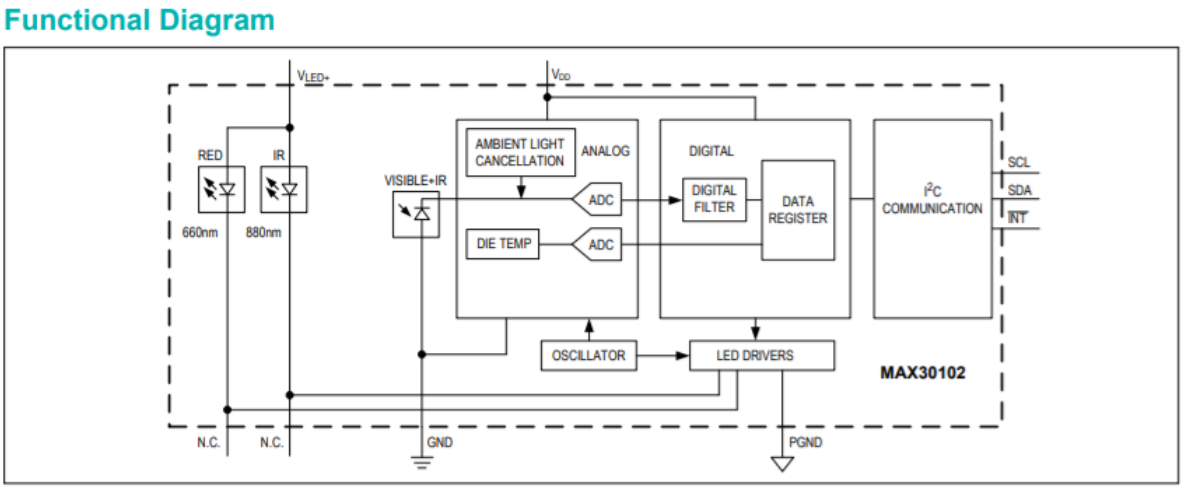



Figura 6. Diagrama de funcionamiento [5]

La electrónica que se encarga de medir la señal eléctrica generada por el fotodiodo se basa en un conversor analógico-digital tipo *sigma-delta continuo* de 18 bits con una frecuencia de muestreo de 1024 MHz. Una vez muestreada la señal, pasa por un pequeño filtrado digital para después ser almacenada en un registro de datos, al que se puede acceder mediante la comunicación i2c. La intensidad de los leds rojo e infrarrojo es programable desde 0 a 50 mA.

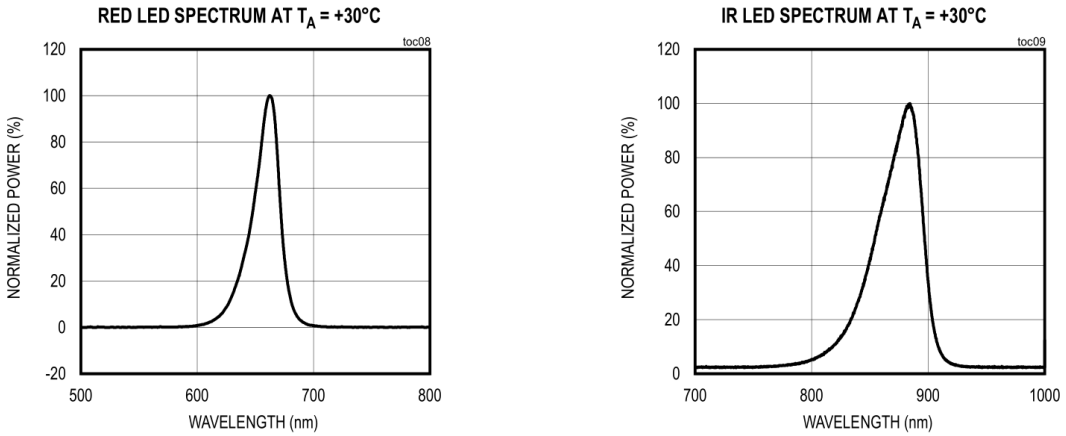



Figura 7. Potencia normalizada vs longitud de onda de los leds rojo e infrarrojo [6]

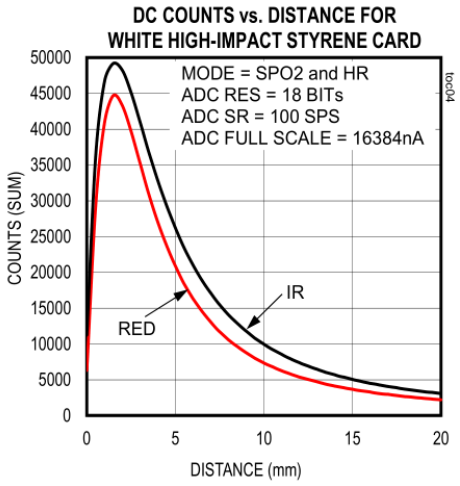



Figura 8. pico de muestras máximas posibles vs distancia [7]

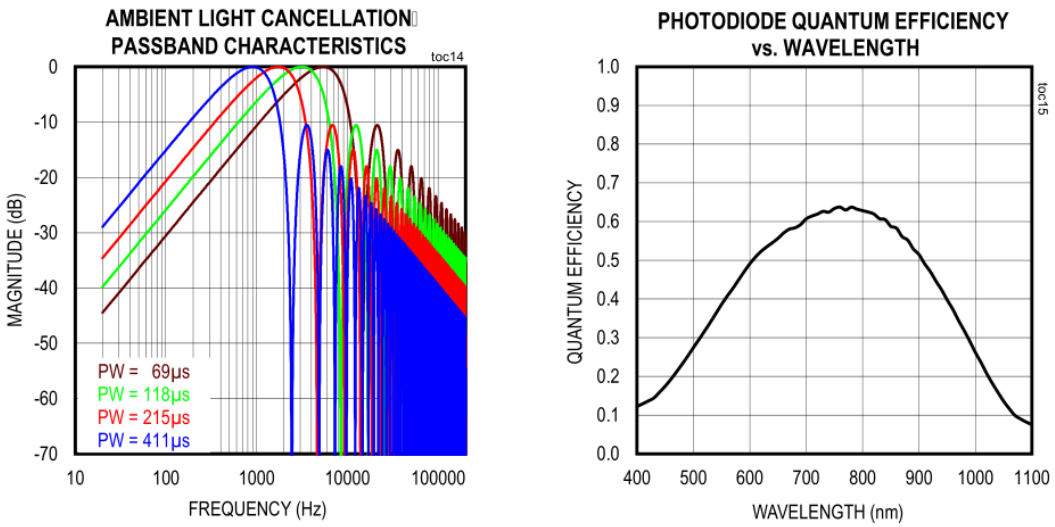



Figura 9. Características físicas del cancelador de luz ambiente y del fotodetector [8]

## 4.3. Conexiones físicas

La placa base del sistema, la ***Mini-KD2***, se conecta a la pantalla mediante sus conectores específicos empleando el bus de datos ***SPI*** y el **puerto paralelo** para la conexión con la pantalla *LCD*. Por otro lado, se conecta la placa con el PC vía *USB* para alimentar la placa y además para poder recibir información vía puerto serie. El sensor se conecta con la placa mediante la comunicación *i2c*. El buzzer se conecta con la placa empleando un pin de propósito general (GPIO). El depurador ***ULINKpro*** se conecta con la ***Mini-KD2*** mediante el conector específico ***JTAG-SW***. Todas las conexiones del sistema están reflejadas en la tabla 1.

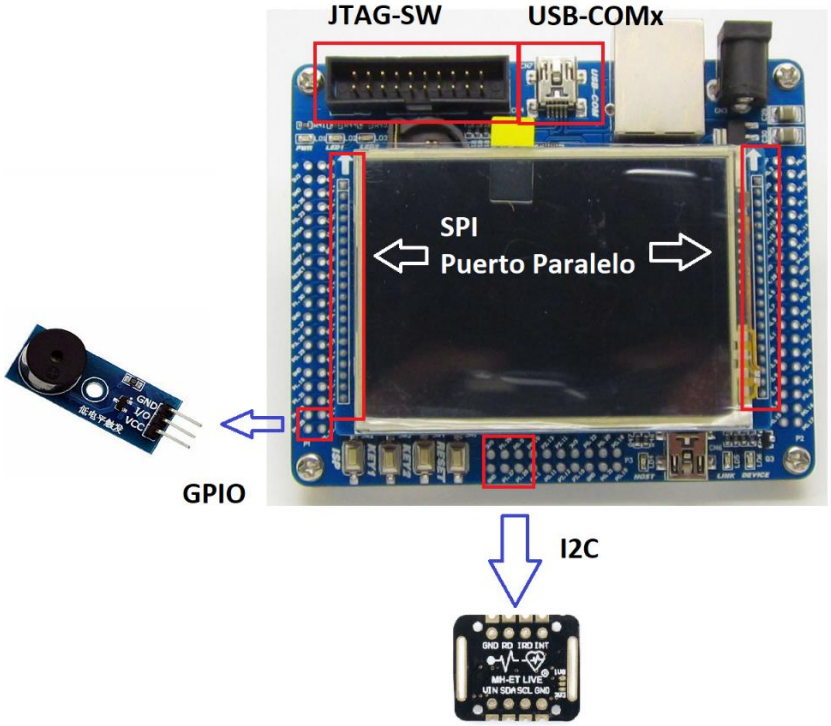



Figura 10. Conexión del sensor con la tarjeta de desarrollo

Tabla 1. Configuración de los pines de la tarjeta mini-dk2 con el sensor y la pantalla

| TARJETA MINI-DK2 | SENSOR MAX30102 | PANEL LCD | PANEL TACTIL | PC | BUZZER |
|---|---|---|---|---|---|
| P0.0 | SDA (gris) | | | | |
| P0.1 | SCL (blanco) | | | | |
| P0.6 a P0.9 y P2.13 | | | SPI | | |
| P1.26 a P1.29 | | PUERTO PARALELO | | | |
| P2.0 a P2.8 | | PUERTO PARALELO | | | |
| P0.15 a P0.22 | | PUERTO PARALELO | | | |
| P0.2 y P0.3 | | | | USB-COMx | |
| CONECTOR 4 | | | | JTAG-SW | |
| 3V3 | VCC (morado) | | | | VCC |
| GND | GND (negro) | | | | GND |
| P1.24 | | | | | GPIO |

## 4.4. Módulo de depuración del LPC1768

El microcontrolador posee un módulo de traza con su correspondiente puerto y un módulo para Test/Depuración con una interfaz **JTAG**. Ambas partes están remarcadas en un recuadro rojo en la figura 12. El módulo de Test/Depuración puede conectarse con la interfaz de depuración estándar **JTAG** de 10 pines. Por otro lado, tiene la opción de Depuración Serie (**SWD**) empleando únicamente 2 pines. La conexión física para interfaz **JTAG** [4] entre el depurador y el microcontrolador se ilustran en la figura 11 .

El microcontrolador tiene una Macrocélula de Traza Embebida (**Embedded Trace Macrocell**) que lo capacita para el seguimiento de instrucciones. **ETM** facilita la depuración para procesadores ARM. Captura información del procesador sin afectar el comportamiento de este. La información de la traza es exportada inmediatamente después de ser capturada a través de un puerto especial. Los microcontroladores que poseen **ETM** permiten que el PC pueda crear una copia virtual de la ejecución de instrucciones del procesador. Esta información puede ser usada para analizar el flujo del programa y localizar errores de software que de otra manera serían difíciles de localizar. Por ejemplo, si el programa se salta inseperadamente alguna instrucción, o si se bloquea en una rutina de interrupción. El depurador provee al usuario los datos de la traza almacenada. El depurador muestra todas las funciones de **ETM** y muestra la información de la traza capturada.

Por otro lado, el microcontrolador tiene incluido una Macrocélula para seguimiento instrumentado (**Instrumented Trace Macrocell [5]** ) de manera que se puede crear un canal y escribir por software en dicho canal para exportar la transmisión de datos de traza, realizado físicamente a través del pin **TDO/SWO** como se muestra en la figura 11.

El **ITM** tiene 32 canales de comunicación, se reparten en 4 grupos de 8 canales, cada grupo puede ser configurable y accesible en modo no privilegiado. El primero y el último canal son exclusivos para la capa CMSIS:

Canal 0: implementa la función **ITM_SendChar** que puede ser usada a modo de imprimir caracteres vía interfaz de depuración.

Canal 31: reservado para el kernel RTOS y puede ser usado para depurar la parte baja del kernel. Sólo puede acceder en modo privilegiado por el propio kenel del RTOS. En algunas ocasiones el propio kernel del RTOS hace uso del canal 31 porque algunos kernels usan el modo privilegiado para la ejecución del programa.

El resto de los canales pueden ser utilizados por el usuario.

En el proyecto he hecho uso del **ITM** como módulo de depuración.

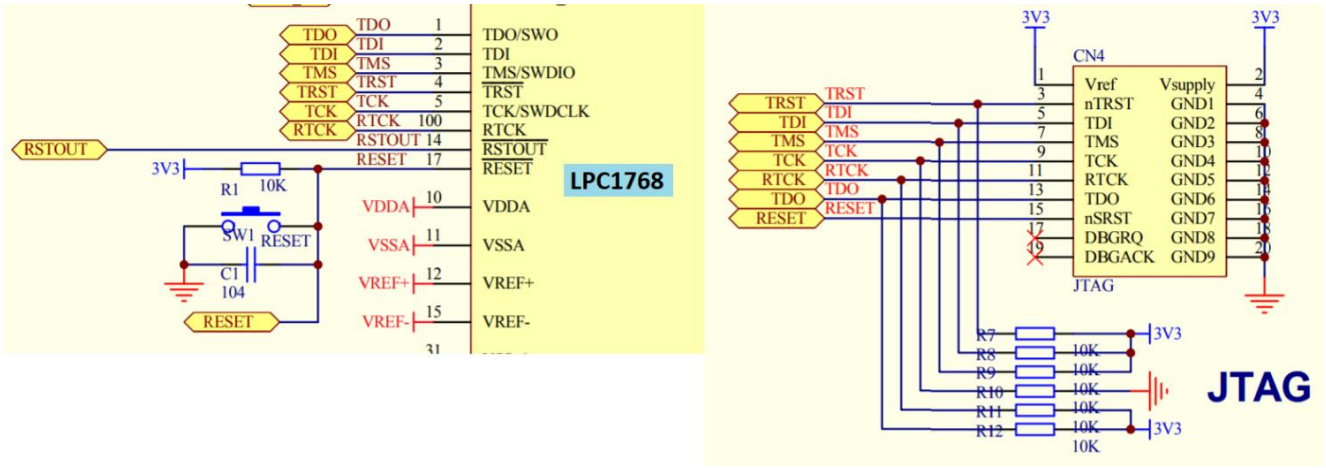



Figura 11. Conexión entre depurador y microcontrolador [9]

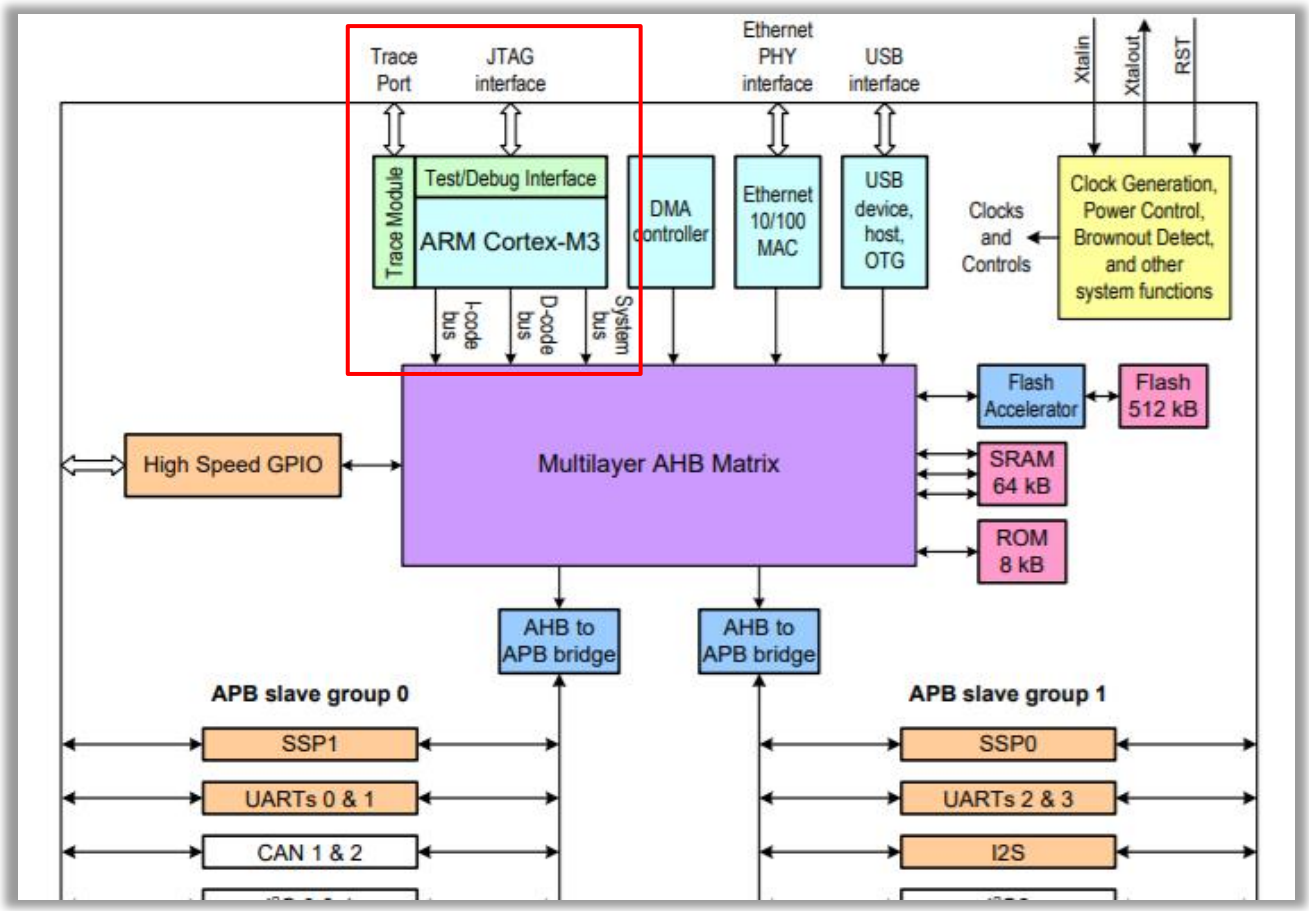



Figura 12. Diagrama de bloques simplificado del LPC17xx [10]

## 4.5. Programador y depurador ULINKpro

Unidad que conecta el PC con la tarjeta del sistema embebido vía **JTAG**, puerto de depuración Córtex, o conector ETM. Esto permite cargar el programa en la memoria **flash** de la tarjeta, acceder a depuración y analizar las aplicaciones usando esta herramienta.

La unidad ***ULINKpro***, junto con el entorno de desarrollo **KEIL** permite al usuario tener el control del procesador, colocar ***breakpoints***, leer y escribir contenido en memoria, todo esto mientras el procesador está trabajando a máxima velocidad. Esta herramienta es muy potente y posee las siguientes características:

- Compatible con dispositivos ARM7, ARM9, Cortex-M0, Cortex-M1, Cortex-M3, y Cortex-M4.
- Compatible con la interfaz JTAG para ATM7, ARM9 y Cortex-M.
- Depuración serie (SWD) para Cortex-M y visor serie (SWV) de datos y eventos para Cortex-M hasta 100MBbit/s.
- ETM para Cortex-M3 y Cortex-M4 hasta 800 Mbit/s.
- Velocidad de hasta 50 MHz para la conexión JTAG.
- Compatible con dispositivos Cortex-M trabajando hasta 200MHz.
- Alta velocidad de lectura y escritura hasta 1Mbyte/s.


Figura 13. Unidad de programación y depuración ULINKpro [11]

# 5. Posibles entornos para desarrollar el software

En el desarrollo del pulsómetro se ha trabajado con un microcontrolador Cortex-M3 de ARM del fabricante NXP y se ha hecho pequeñas pruebas para evaluar diferentes entornos de desarrollo.

A la hora crear un nuevo proyecto a nivel de software como primera opción se ha tenido en cuenta el entorno ***Keil uVision5 [6]*** porque es el que se ha empleado durante el grado y porque tiene un recurso verdaderamente útil llamado *Manage Run-Time Environment*. Otra opción es hacer uso del entorno gráfico que proporciona ***STM32CubeMX***. También se puede emplear el entorno ***Eclipse*** o se puede trabajar desde la nube empleando la aplicación ***MBed*** de ARM.

La ventaja de emplear la aplicación ***STM32CubeMX*** es la facilidad con que se pueden inicializar los periféricos de manera gráfica lo que ahorra mucho tiempo al desarrollador. El único inconveniente es que sólo se puede trabajar con microcontroladores de la familia STM32. El entorno ***Eclipse*** es muy parecido al ***STM32CubeMX,*** pero en sus versiones anteriores y no posee herramienta gráfica a la hora de inicializar los proyectos. Por otro lado, trabajar con ***MBed*** tiene la ventaja de crear proyectos rápidamente y una comunidad muy grande que expone sus proyectos de manera libre, pero el inconveniente es que está limitado a las tarjetas con las que comercializan. En cambio, si se hace uso del entorno de desarrollo ***Keil uVision5*** se puede trabajar con una amplia gama de microcontroladores de diferentes familias, posee un entorno para añadir periféricos al proyecto y es muy versátil si se quiere tener el control del código en todo momento, además del potente recurso de depuración que posee.

Para depurar el sistema del proyecto se empleará la herramienta *Tracealycer* de *Percepio* donde se pueden visualizar el uso de la CPU, los eventos del sistema o el comportamiento de las tareas al interrumpirse unas a otras según la prioridad.

# 6. Sistema Operativo de Tiempo Real FreeRTOS

El sistema operativo ***FreeRTOS [7]*** es un kernel de tiempo real de software libre con énfasis en la facilidad de su uso. Originalmente desarrollado por Richad Barry en 2003, desarrollado junto a varias compañías líderes mundiales durante 18 años, actualmente lo administra Amazon (*Amazon Web Services*). Soporta hasta 40 arquitecturas, más de 15 *toolchains* y actualmente es descargado cada 170 segundos. Además del kernel, existe un set de librerías en continuo desarrollo adecuadas para todos los sectores de la industria.


Figura 14. Sistema Operativo de tiempo real FreeRTOS [12]

Tanto el *kernel* como las librerías se distribuyen bajo la licencia de código abierto del MIT. Dicha licencia permite utilizar el software sin restricciones incluyendo los derechos de uso, copia, modificación, fusión, publicación, distribución del software bajo una serie de condiciones que se especifican en su propia página en el apartado *licencia*.

Por otra parte, se ha creado ***OpenRTOS***, una versión con licencia comercial del *kernel* ***FreeRTOS*** que incluye soporte dedicado. ***OpenRTOS*** se proporciona bajo la licencia de AWS por WITTENSTEIN High Integrity Systems, un socio estratégico de AWS. A modo de comparación entre la licencia MIT y la comercial de ***FreeRTOS*** se ha creado la tabla 2.

Tabla 2. Comparación de licencias del kernel de FreeRTOS

| | Licencia MIT del kernel FreeRTOS | Licencia comercial OpenRTOS |
|---|---|---|
| ¿Es gratis? | sí | No |
| ¿Puedo usarlo en una aplicación comercial? | sí | sí |
| ¿Se puede utilizar sin necesidad de pagar regalías o derechos de licencia por cada uso? | sí | sí |
| ¿Se proporciona garantía? | No | **sí** |
| ¿Puedo recibir soporte técnico profesional sobre una base comercial? | sí | sí |
| ¿Se proporciona protección legal? | No | **Sí** , se proporciona protección contra la infracción de la propiedad intelectual |
| ¿Tengo que convertir en software libre el código de mi aplicación que utiliza los servicios FreeRTOS? | No | No |
| ¿Tengo que convertir en software libre el código fuente de mis cambios en el kernel de FreeRTOS? | No | No |
| ¿Tengo que documentar que mi producto utiliza FreeRTOS? | No | No |
| ¿Tengo que proporcionar el código FreeRTOS a los usuarios de mi aplicación? | No | No |

## 6.1. Características principales

- Sistema multitarea: es capaz de ejecutar varios procesos aparentemente al mismo tiempo, compartiendo uno o más procesadores.
- Mínimo tamaño de memoria ROM y RAM, la imagen del ***kernel*** ocupa típicamente un tamaño de 6KB a 12 KB .
- Organización *microkernel*: Algunas funciones del SO se implementan como tareas similares a las de la aplicación.
- Configurable / escalable: Se puede compilar solo las funcionalidades necesarias.
- *Timers* muy eficientes propios del SO: por software, no hacen uso de los recursos del Hardware.
- Portable: nunca se desactivan por completo las interrupciones facilitando el uso de la API.
- Código fuente en C de muy alta calidad: bajo una estricta gestión de configuración.
- Amplia documentación para el aprendizaje y la práctica de ingenieros.
- Ejemplos de proyectos pre-configurados para soportar portabilidad.
- Soporte gratuito, mejor valorado que otras alternativas comerciales.

- Comunidad extensa y en crecimiento.
- Solución completa, robusta, libre de riesgos, y de bajo coste.
- Colas, semáforos binarios y de recuento, mutex y mutex recursivos.
- Referencia para la integración de IoT por tener muchos ejemplos y librerías esenciales para conectarse con seguridad a la nube.
- Licencia y soporte comerciales opcionales.

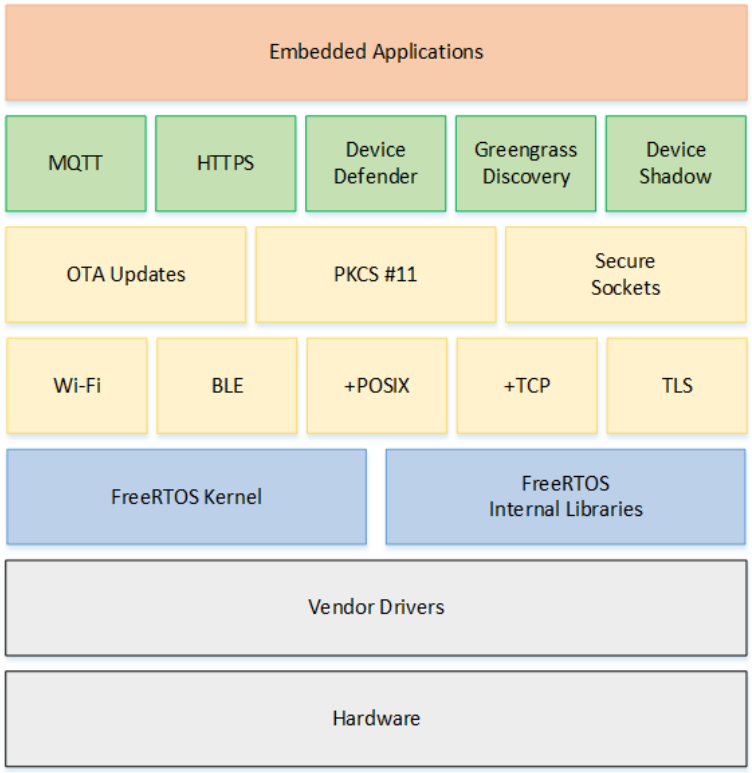



Figura 15. Diagrama de Bloques de los distintos niveles de un sistema embebido de AWS Amazon FreeRTOS [13]

## 6.2. Modelo de programación

Una de las principales ventajas de usar un S.O. es la capacidad que tiene el núcleo de ejecutar múltiples tareas, es decir, una tarea puede suspender su ejecución para que otra se ejecute sin ningún problema, en periodos cortos de tiempo, dando la impresión de que las tareas se ejecutan a la vez, por lo que aumenta la confiabilidad del sistema.

En primer lugar, se deben crear las tareas, y a continuación inicializar el planificador. Si la CPU se compone de un solo núcleo, el sistema únicamente puede ejecutar una única tarea dejando las otras en estado de espera. En la figura 16 se ilustra este comportamiento a lo largo del tiempo. Si la CPU tiene varios núcleos las tareas que necesiten ser ejecutadas se reparten entre ellas.

Del mismo modo si el sistema es de un único núcleo y se requiere ejecutar una interrupción o algún evento del sistema, dejará en estado de suspensión la tarea que esté ejecutando y pasará a atender la interrupción o el evento.

En este proyecto se ha empelado una asignación de prioridades a las distintas tareas para tener un comportamiento excluyente, es decir, que las tareas de mayor prioridad desalojarán a las de menor prioridad.

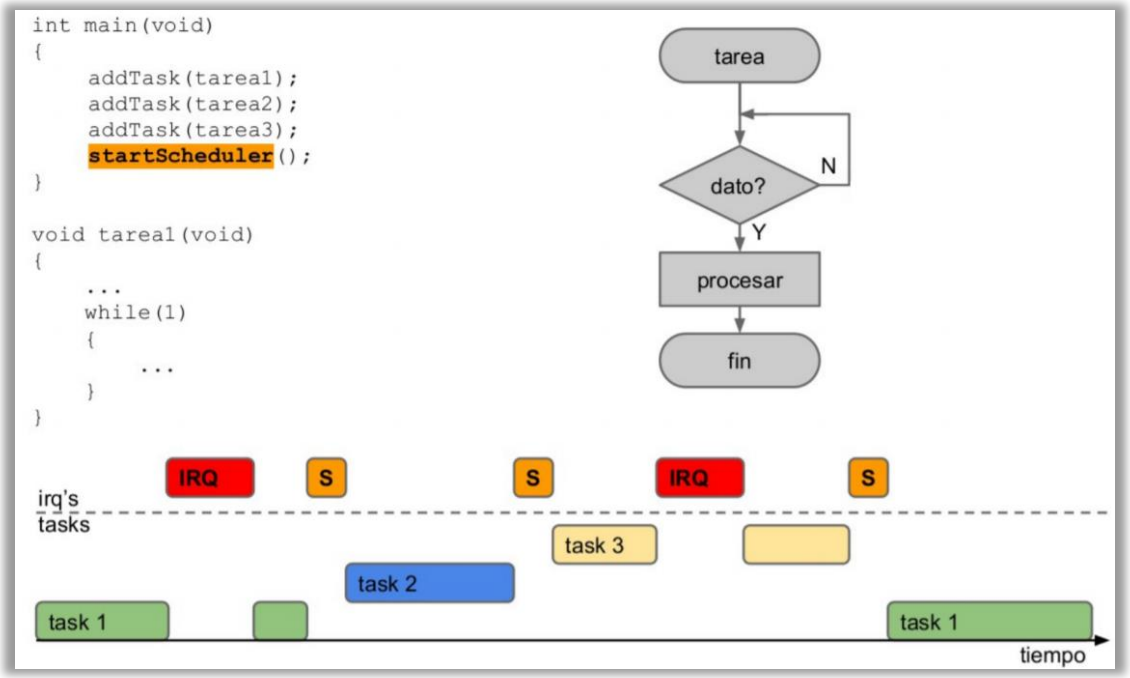



Figura 16. Modelo de programación [14]

## 6.3. Creacion de un proyecto bajo FreeRTOS

La aplicación de ***FreeRTOS*** no se inicia hasta que se llame a la función de inicialización del planificador *" vTaskStartScheduler()"* ,que requiere de los siguientes ficheros[10] [11] :

- *FreeRTOS/Source/tasks.c*
- *FreeRTOS/Source/queue.c*
- *FreeRTOS/Source/list.c*
- *FreeRTOS/Source/portable/[compiler]/[architecture]/port.c*
- *FreeRTOS/Source/portable/MemMAng/heap_x.c* (donde "x" es 1, 2 , 3, 4 o 5)

Opcionalmente se pueden añadir los ficheros de timer por software FreeRTOS/Source/timers.c , los de grupos de eventos, de manejo de buffers para la comunicación entre tareas, o las de co-rutinas.

También se deben añadir los ficheros de cabecera que se encuentran en los siguientes directorios:

- *FreeRTOS/Source/include*
- *FreeRTOS/Source/portable/[compiler]/[architecture]*

## 6.4. Comportamiento de una tarea

Las tareas pueden tener cuatro distintos estados que quedan ilustrados en la siguiente figura y que se detallarán a continuación.

- **Running** o **Ejecutándose**. La tarea tiene asignado un núcleo de la CPU y se está ejecutando.
- **Ready** o **Preparada**. La tarea está habilitada para ser ejecutada, pero no lo hace porque existe otra tarea de igual o de mayor prioridad que se está ejecutando. Una vez que se ha creado una tarea, está lista para ejecutarse.
- **Bloqueada**. El sistema necesita ejecutar una interrupción o algún evento del sistema.
- **Suspendida**. Similar al estado de bloqueo con la diferencia de que solo se puede acceder o salir de este estado cuando el programador llame a las funciones *VtaskSuspend() o vTaskResume()* respectivamente.

Cada tarea consume una cantidad de memoria *RAM* que se asigna dinámicamente al crearla. Concretamente corresponde con la zona de memoria *Heap* del sistema operativo. La función para crear una tarea recibe como parámetros:

- Nombre de la función que realmente es la tarea.
- Nombre de la tarea que se mostrará en depuración.
- Tamaño mínimo de la memoria de la tarea en la pila.
- Prioridad que se asigna a la tarea propia.
- Prioridad de recepción de cola. No se usará en el proyecto, por tanto, dicho valor será *NULL*.

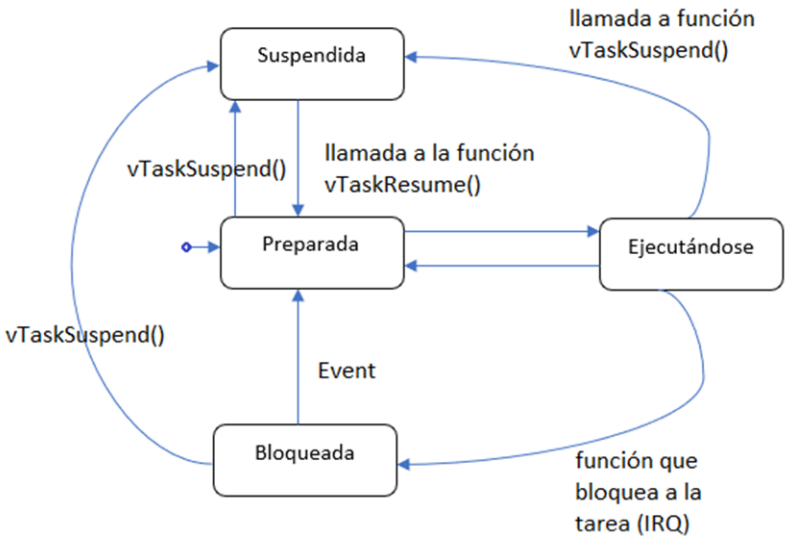



Figura 17. Distintos estados de una tarea y posibles transiciones entre ellas

## 6.5. Descripción de las tareas del sistema embebido

El sistema se compone principalmente del sensor oxímetro de pulso *MAX30102* que está conectado a la tarjeta de desarrollo M*ini-DK2* a través del puerto *i2c*. Por otro lado, se conecta la M*ini-DK2* al PC mediante el *puerto USB-COM* de la tarjeta de desarrollo. Para diferenciar cuando se ejecuta una tarea u otra, se ha empleado el **LED1** para indicar que el sistema ejecuta la tarea 1, y el **LED2** para indicar que se ejecuta la tarea 2.

La aplicación desarrollada para el microcontrolador *LPC1768* sobre el sistema operativo ***FreeRTOS*** inicializa y recibe los datos del oxímetro de pulso para hacer los cálculos correspondientes y mostrar en la pantalla del sistema el nivel de oxígeno en sangre y las pulsaciones por minuto. En caso de que hubiera algún tipo de problema con la pantalla, se pueden visualizar los datos en cualquier aplicación de comunicaciones puerto serie para PC *COMx* conectando la *mini-DK2* al *PC* a través del puerto *USB-COM*.

Para entender el análisis que se realice al sistema, es importante saber cómo tienen que comportarse las tareas previamente, cómo se han creado y configurado.

El sistema diseñado consta de 2 tareas repartiéndose el trabajo y configurándose de la siguiente manera. Se recuerda que cuando se trabaja con el sistema operativo ***FreeRTOS*** cuanto más alto es el valor de la prioridad asignada a una tarea, más prioritaria es. Se ha creado la tarea 1 en primer lugar, la tarea 2 en segundo lugar. A continuación de crear las tareas, se suspende inmediatamente la tarea 2 por software, de manera que la primera tarea que debe ejecutarse inmediatamente es la tarea 1.

a) Tarea 1

Se ha creado con el nombre de *"**Calcula_Oxigeno**"*, con prioridad ***tskIDLE_PRIORITY* + 2** con un tamaño reservado en pila de **128** bytes tal y como se indica en la definición ***configMINIMAL_STACK_SIZE*** del fichero ***FreeRTOSConfig.h***. Esta tarea se encarga de inicializar el sensor y recibir las muestras obtenidas vía i2c, procesar las muestras de nivel de oxígeno en sangre y calcular el nivel de oxígeno en sangre en tanto por ciento. Además, detecta si el usuario tiene el dedo sobre el sensor, en caso contrario envía un mensaje indicando que se coloque el dedo sobre el sensor. Una vez que se calcula el nivel de oxígeno, se pasa a ejecutar la tarea 2.

b) Tarea 2

Se ha creado con el nombre de *"**Calcula_Pulsaciones**"*, con prioridad ***tskIDLE_PRIORITY* + 3** con un tamaño reservado en pila de **128** bytes tal y como se indica en la definición ***configMINIMAL_STACK_SIZE*** del fichero ***FreeRTOSConfig.h***. Esta tarea se encarga de configurar el sensor para recibir las muestras que indican las pulsaciones del usuario, y realiza los cálculos necesarios para obtener las pulsaciones por minuto. Calcula continuamente las pulsaciones por minuto mostrándola por la pantalla LCD y por el puerto serie. Si detecta que el usuario no tiene el dedo sobre el sensor, envía un mensaje indicando que se coloque el dedo sobre el sensor. Una vez que se vuelve a introducir el dedo sobre el sensor, el sistema pasa a ejecutar nuevamente la tarea 1.

c) Tarea *Tmr Svc*

Tarea que la crea el propio sistema operativo que se observa en depuración y que consiste en una tarea que mantiene ordenada la lista de timers por software que se hayan creado por el sistema operativo. Esta tarea se crea al iniciar el planificador y queda en estado suspendida porque en el proyecto no se ha hecho uso de ningún timer por software sobre ***FreeRTOS***.

Tabla 3. Descripción de las tareas del sistema embebido

| Tarea | Nombre | prioridad | tamaño en pila | Funciones |
|---|---|---|---|---|
| Tarea 1 | Calcula_Oxigeno | 2 | 128 | Inicializar sensor<br>Leer FIFO<br>Calcular nivel de oxígeno<br>actualizar hora<br>detección de presencia sobre el sensor |
| Tarea 2 | Calcula_Pulsos | 3 | 128 | Inicializar sensor<br>Leer FIFO<br>Calcular pulsos por minuto<br>actualizar hora<br>detección de presencia sobre el sensor |

Se ha modelado el comportamiento del sistema con la máquina de estados de la figura 18.

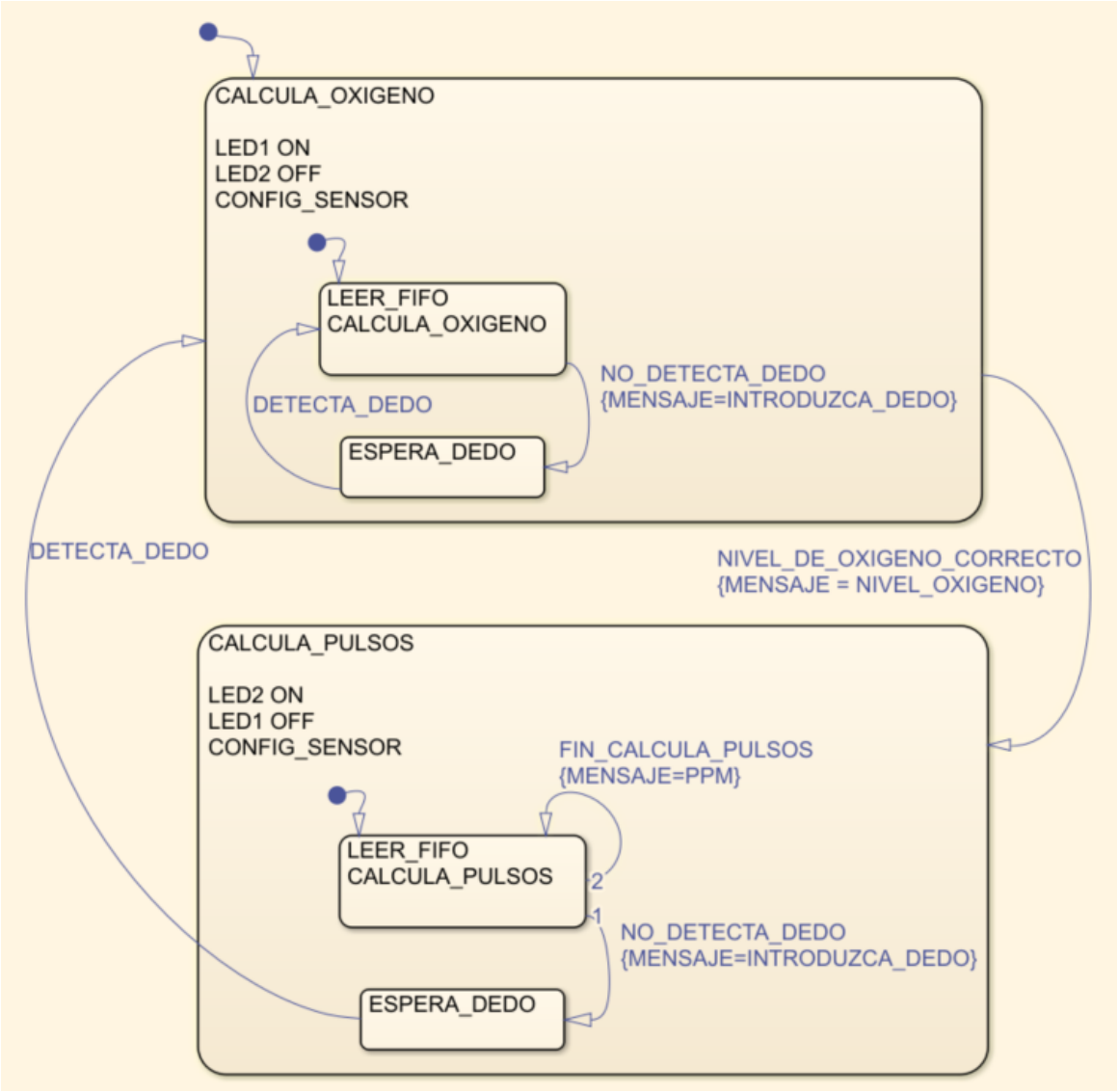



Figura 18. Máquina de estados del sistema

## 6.6. Configuración del sensor MAX30102

En este apartado se detalla la configuración del sensor MAX30102 y se explica cómo se realizan los cálculos para obtener el nivel de oxígeno en sangre y las pulsaciones por minuto.

Para un correcto funcionamiento del sensor se ha adaptado la librería ***SparkFun_MAX3010x_Sensor_Library*** al microcontrolador LPC1768.

Para calcular el nivel de oxígeno, se debe configurar el sensor de la siguiente manera:

Tabla 4. Registros de configuración del ***MAX30102***

| Registro | Dirección | Valor | Función |
|---|---|---|---|
| Mode Configuration | 0x09 [2:0] | 2 | Calcula nivel de oxígeno, LED IR y ROJO activos |
| LED Pulse Amplitude | 0x0C-0x0D | 60 | Corresponde a 11,76 mA (Valores entre 0 y 255). |
| SampleAverage | 0x08 | 4 | Para reducir la cantidad de datos a transferir, muestras adyacentes pueden promediarse y diezmarse en el chip |
| SampleRate | 0x0A [4:2] | 1 | Define un muestreo de 100 muestras por segundo. Cada muestreo consiste en una medida de luz IR y otra de luz ROJA |
| PulseWidth | 0x0A [1:0] | 3 | Duración de 411 us del LED en estado activo durante el muestreo |
| adcRange | 0x0A [6:5] | 1 | Corriente máxima que el ADC es capaz de medir de 4096 nA |

Los 3 bits más bajos del registro ***Mode Configuration*** indican el modo de funcionamiento. Según la tabla 4, se ha configurado el modo $\boldsymbol{SpO_2}$, esto quiere decir que se va a calcular el oxígeno en sangre, activando los leds Rojo e Infrarrojo.

La cantidad de fotones generados es directamente proporcional a la corriente que circula por los leds por lo que se configura el registro ***LED Pulse Amplitude*** con un valor de 60, esto quiere decir que

$i_{diodo} = \frac{LED_{Pulse_{Amplitude}}}{255}$, con un valor de 60 se obtiene una corriente de 11,76 mA a través de los leds Rojo e Infrarrojo.

Para reducir la cantidad de datos a transferir entre el sensor y el microcontrolador LPC1768, el propio sensor tiene la opción de realizar un promedio según el valor del registro ***SampleAverage***. Se ha configurado un promediado cada 4 muestras.

El registro ***SampleRate*** indica el número de muestras por segundo que el sensor va a realizar. Si se trata de muestrear la luz roja e infrarroja, como es en el caso del modo $\boldsymbol{SpO_2}$, una muestra implica realizar una medida de luz roja y otra de luz infrarroja.

El ADC interno del sensor tiene una resolución desde 15 bits hasta 18 bits configurables mediante el registro ***adcRange***.

Los leds rojo e infrarrojo no pueden estar activos al mismo tiempo, se trabaja mediante pulsos cortos alternados, y existe un periodo de tiempo de separación entre el pulso del led rojo y el pulso del led infrarrojo. Este tiempo de separación limita la frecuencia máxima de muestreo a la que puede trabajar el sensor.

La duración del pulso de cada led determina la resolución del ADC de manera que 69 us significa que se trabaja con el ADC a 15 bits, 118 us – 16 bits, 215 us – 17 bits y 441 us – 18 bits (ver figura 20).

El tiempo de separación entre pulsos lo especifica el fabricante y se puede ver en la figura 19. Se ha establecido un ancho de pulsos de 411 us de manera que existe una separación de 696 us entre pulsos, lo que limita la frecuencia de muestreo a $f_{muestreo} = \frac{1}{411\ us + 696\ us} = 903\ muestras\ por\ segundo$ en el caso de trabajar en el modo ***HR Mode***, modo medidor de pulsos, aunque en el fabricante indica que se podría trabajar hasta 1000 muestras por segundo, podría no ser seguro.

Por otro lado, si se trabaja en ***SpO$_2$ Mode***, medidor del oxígeno en sangre, se trabajan con dos leds, por lo que la frecuencia máxima de trabajo con un ancho de pulso de 411 us, es de 400 muestras por segundo.

| PULSE-WIDTH SETTING (µs) | CHANNEL SLOT TIMING (TIMING PERIOD BETWEEN PULSES) (µs) | CHANNEL-CHANNEL TIMING (RISING EDGE-TO-RISING EDGE) (µs) |
|---|---|---|
| 69 | 358 | 427 |
| 118 | 407 | 525 |
| 215 | 505 | 720 |
| 411 | 696 | 1107 |

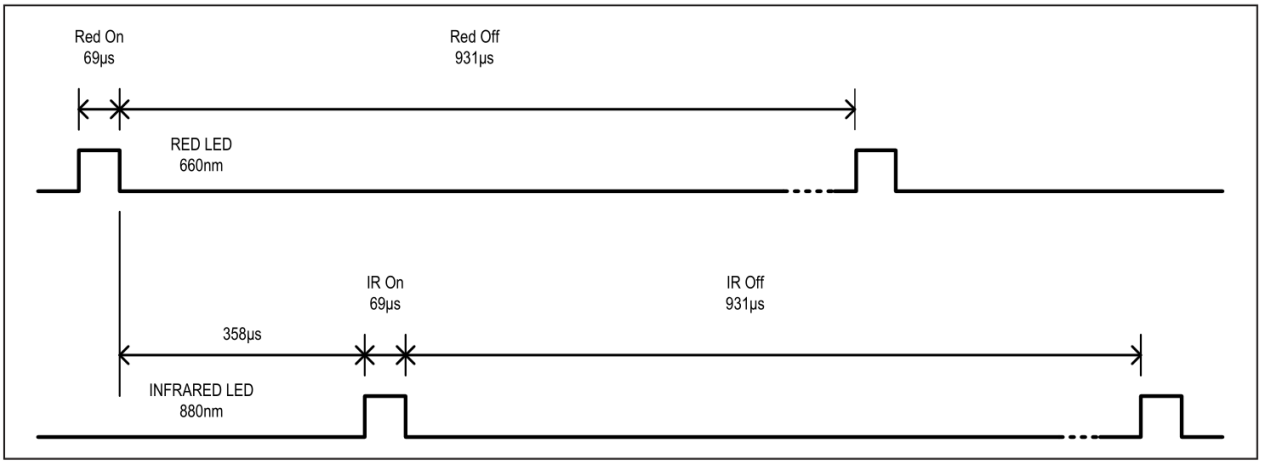



*Figure 3. Channel Slot Timing for the SpO$_2$ Mode with a 1kHz Sample Rate*

Figura 19. Tiempos durante la adquisición de una muestra [15]

**Table 11. SpO$_2$ Mode (Allowed Settings)**

| SAMPLES PER SECOND | PULSE WIDTH (µs) | | | |
|---|---|---|---|---|
| | 69 | 118 | 215 | 411 |
| 50 | O | O | O | O |
| 100 | O | O | O | O |
| 200 | O | O | O | O |
| 400 | O | O | O | O |
| 800 | O | O | O | |
| 1000 | O | O | | |
| 1600 | O | | | |
| 3200 | | | | |
| Resolution (bits) | 15 | 16 | 17 | 18 |

**Table 12. HR Mode (Allowed Settings)**

| SAMPLES PER SECOND | PULSE WIDTH (µs) | | | |
|---|---|---|---|---|
| | 69 | 118 | 215 | 411 |
| 50 | O | O | O | O |
| 100 | O | O | O | O |
| 200 | O | O | O | O |
| 400 | O | O | O | O |
| 800 | O | O | O | O |
| 1000 | O | O | O | O |
| 1600 | O | O | O | |
| 3200 | O | | | |
| Resolution (bits) | 15 | 16 | 17 | 18 |

Figura 20. Tablas con las frecuencias posibles según el ancho de pulso configurado [16]

# 7. Introducción a Percepio Tracealyzer

***Percepio Tracealyzer*** es una solución novedosa por ser la primera en cuanto al diagnosis de la traza de datos de manera visual. Permite depurar fácilmente los errores de niveles de sistema, encontrar defectos de diseño de software y medir los tiempos de sincronización del software y el uso de recursos. Esto asegura que el código sea fiable, eficiente y eficaz.


Figura 21. Logotipo de Percepio Tracealyzer [17]

Este recurso software puede ser usado con depuradores tradicionales como con los entornos IAR, Keil o Eclipse, y completar la vista de depuración detallada con vistas adicionales de niveles del sistema, ideal para entender fallos de tiempo real donde un depurador clásico no sería capaz. Por tanto, es muy útil a la hora de depurar un sistema de tiempo real como por ejemplo el control de un motor, porque no es posible parar el sistema para depurarlo. Empleando este software de depuración, es posible grabar la ejecución del software del sistema embebido y algunas variables de interés en tiempo real mientras el sistema se ejecuta.

Richard Barry, fundador de ***FreeRTOS*** citó - "***Tracealyzer*** siempre ha ofrecido un valor excepcional a nuestra comunidad global de usuarios al brindar a los ingenieros información directa sobre cómo se ejecutan sus aplicaciones. Esta información es de gran interés al optimizar y depurar aplicaciones ***FreeRTOS***."

***Tracealyzer*** provee más de 30 puntos de vistas en forma de ventanas para visualizar:

- Comportamientos en tiempo de ejecución
- Planificación de tareas
- Ejecución de interrupciones/ISR
- Tiempos de tareas, prioridades de tareas
- Carga de CPU
- Uso de memoria
- Interacciones entre tareas e interrupciones/ISRs vía cola de mensajes, semáforos y mutex objects. De este modo, se puede ver

- Si una tarea prioriza sobre otra
- si los tiempos de ejecución y los timeouts son los adecuados
- si ocurre algún tipo de error de prioridad o de tiempos de una tarea en específico.

También permite personalizar logs como "Eventos de Usuario" del código de aplicación para ver estados y variables mientras se usan en paralelo otros puntos de vista. Esto puede ser empleado para analizar y depurar el comportamiento de tiempo real de algoritmos, como lazos de control de motores, para ver la lógica del software como su rendimiento. Por ejemplo, el tiempo de ejecución que necesita, si se puede ejecutar el lazo de control en menos tiempo, y si el tiempo de la tarea es constante.

Todos los puntos de vista que ofrece ***Tracealyzer*** están interconectados de modo que se puede clicar en un punto de algún dato en específico para saltar a otra ventana donde se ofrece otro punto de vista.

La traza capturada puede ser transmitida en tiempo real, empleando una sonda de depuración o usando una interfaz *target-host* como *USB* o *TCP/IP*.

No se requiere un recurso hardware especial, de manera que se trabaja con cualquier procesador o microcontrolador asumiendo que se tiene disponibles unos pocos de kilobytes para el módulo que graba la traza.

Habitualmente, los microcontroladores poseen un módulo que se encarga de grabar la traza y generalmente está presente en los diseños de procesadores de 32 bits y los que estén por encima de ellos, incluyendo microcontroladores *Cortex-M de ARM*, los de *STMicroelectronics STM32*, *NXP LPC and Kinetis series*, *Renesas Synergy, Silicon labs EFM32*, *Cypress PSoC, Atmel SAM*, *TI MSP432*, *TI TMC4* y *Infineon XMC4*. También soporta *Renesas RX, Renesas RZ*, *Microchip PIC32, Atmel AVR32*, *ARM Cortex-R, ARM Cortex-A*, *Xilinx Zynq*, *Xilinx Microblaze*, *Altera Nios II* y procesadores *Synopsys ARC*. Si alguna familia de procesadores no está directamente soportada, se puede desarrollar fácilmente su propio puerto. Solo se necesita definir unas pocas macros.

También está presente en sistemas habilitados para ***IoT***, implementado a través de *Percepio DevAlert*, para buscar errores y obtener resultados de manera remota.

Está disponible en un amplio número de entornos de desarrollo destacando:

- *IAR Embedded Workbench*
- *Keil µVision (MDK)*
- *Atmel Studio*
- *Microchip MPLAB X IDE*
- *Wind River Workbench*
- muchos IDEs basados en *Eclipse como Atollic TrueStudio,*
- *SW4STM32*
- *Code Composer Studio (TI CCS)*
- *NXP LPCxpresso/MCUxpresso*

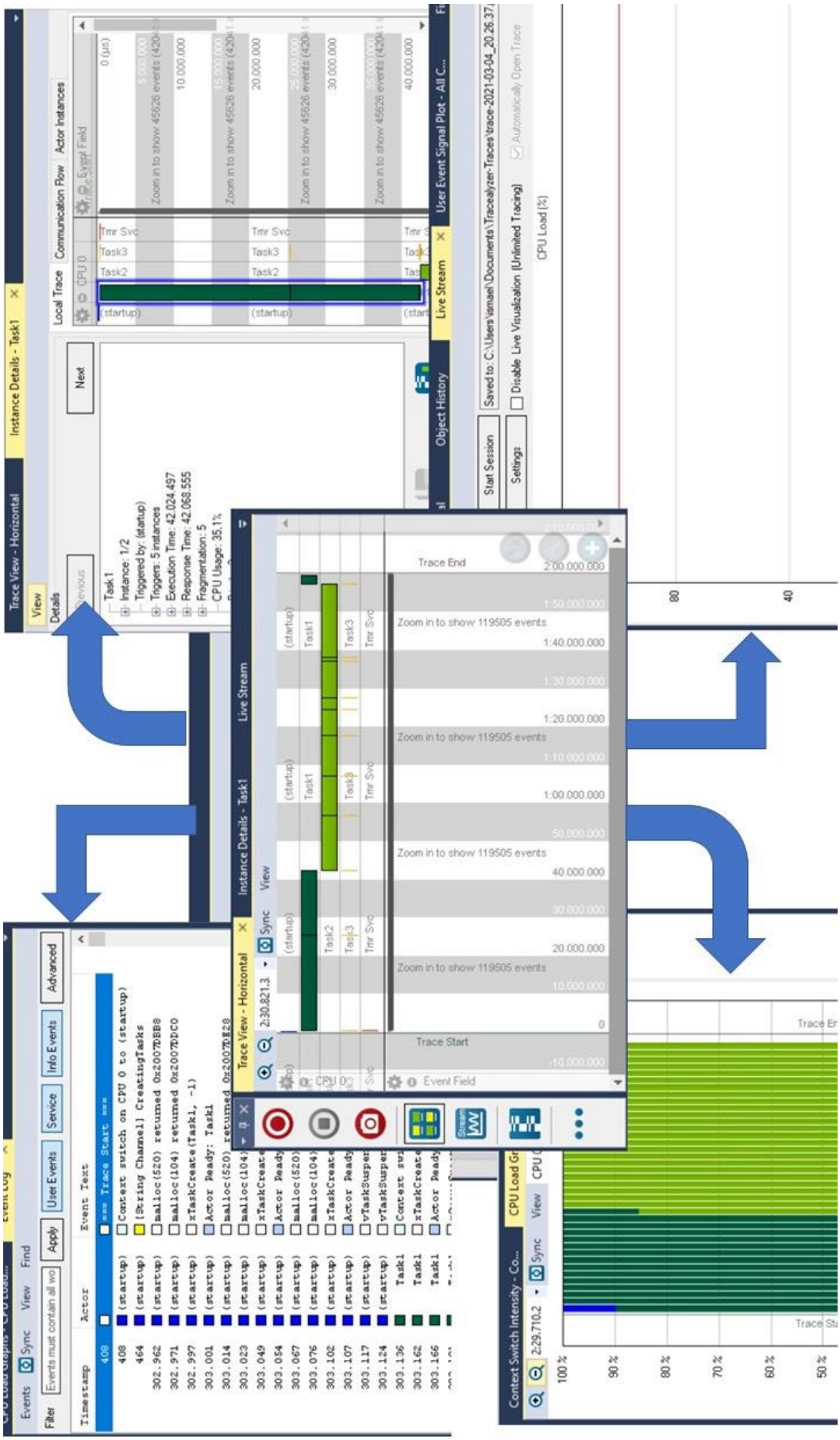



Figura 22. Puntos de vista en formas de ventanas de la herrmaienta Tracealyzer

***Tracealyzer*** puede ser usado en cualquier IDE que sea capaz de almacenar contenido de la memoria RAM en un fichero .bin o .hex, o que posea alguna interfaz *target-host* para la transmisión de la trazabilidad en forma de datos al host PC.

Es compatible con multiples RTOS y plataformas Linux, como *FreeRTOS, Amazon FreeRTOS, Arm Keil RTX5, Wittensein SafeRTOS, Express Logic ThreadX, Micrium uC/OS-iii, On time RTOS-32, OpenVX – Synopsys EV6x, Wind River VxWorks*.

***Percepio Tracealyzer*** consiste en dos componentes:

- La aplicación para PC Tracealyzer, que provee la visualización de la traza. Oficialmente disponible para Windows y Linux.
- Percepio proporciona una librería llamada *Percepio Trace Recorder* compatible con varios RTOS con configuraciones completas. Dicha librería se encarga de crear las trazas en forma de datos y enviarlas al PC host.

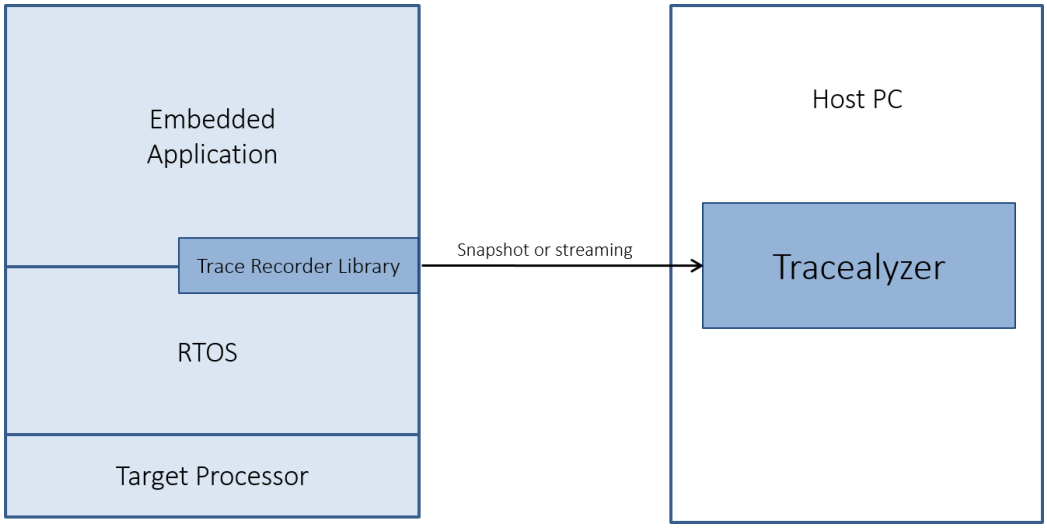



Figura 23. Diagrama de conexión del sistema embebido con el pc host [18]

## 7.1. Creación del programa principal en el entorno KEIL uVision5

Para crear un proyecto se ha hecho uso del entorno ***Keil uVision5*** porque posee una interfaz muy intuitiva para que el desarrollador comience cualquier proyecto en muy poco tiempo. Para comenzar se debe seleccionar el microcontrolador que se desee emplear. A continuación, se debe seleccionar ***FreeRTOS*** desde la API RTOS en la ventana ***Manage Run-Time Environment***. Esta ventana permite seleccionar los periféricos con los que se va a trabajar, la capa CMSIS que se vaya a emplear, el sistema operativo con el que se vaya a trabajar y los recursos esenciales para el proyecto.

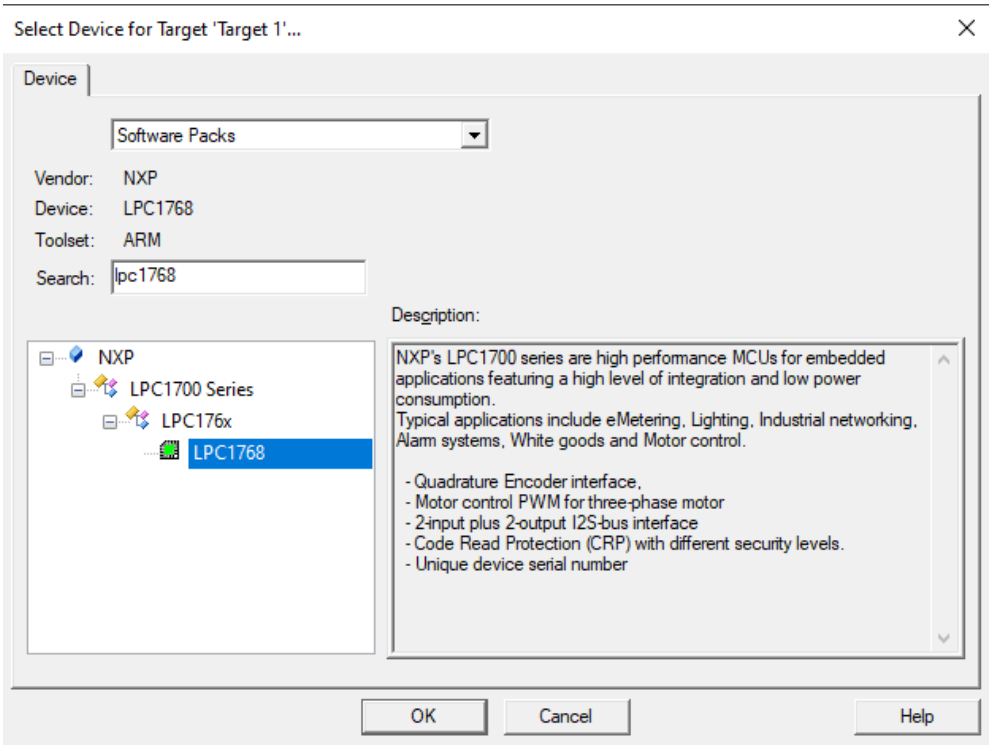



Figura 24. Selección del dispositivo

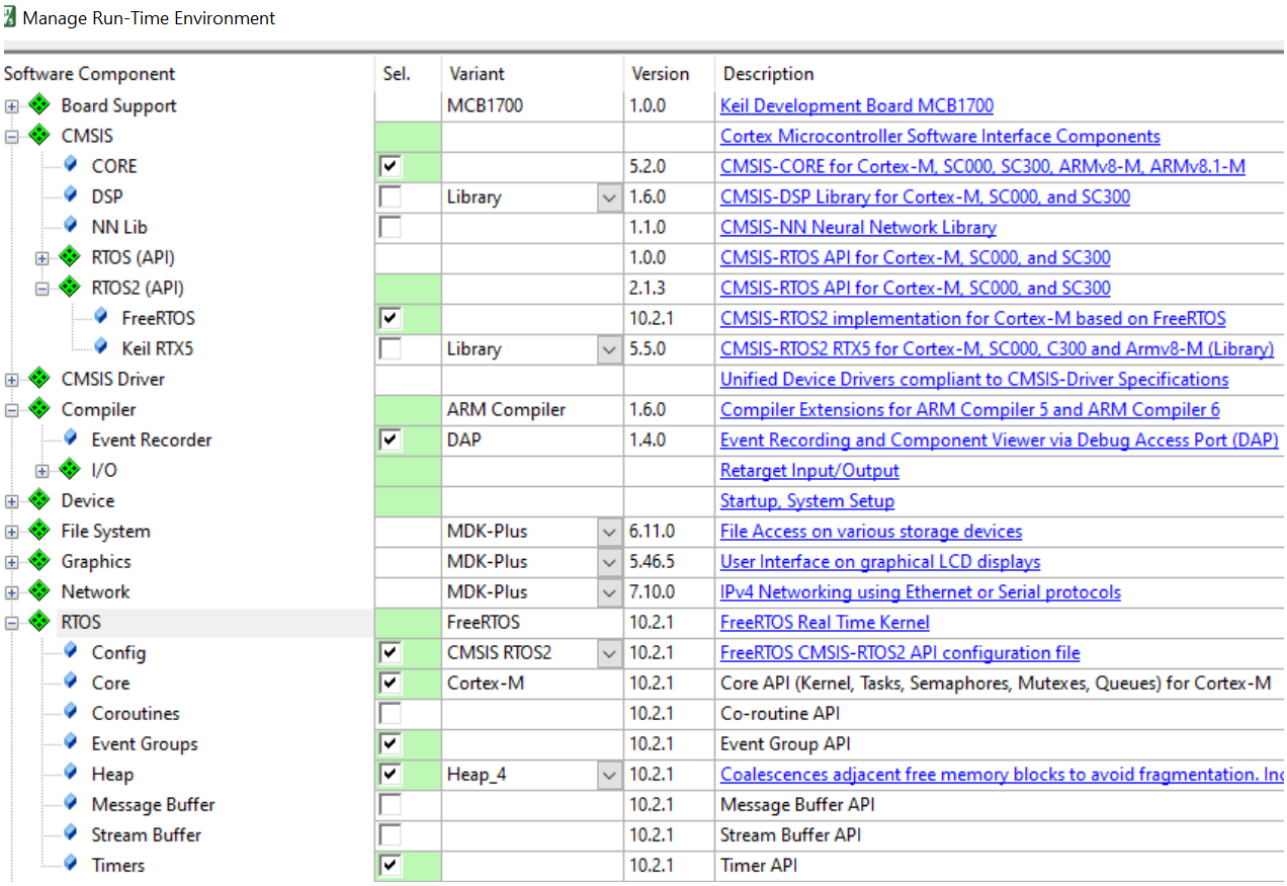



Figura 25. Interfaz de inicialización del proyecto en Keil uVision5

## 7.2. Adaptando el proyecto a Tracealyzer

Una vez se haya habilitado la trazabilidad en la tarjeta del sistema, se puede iniciar y parar de grabar la traza desde la aplicación ***Tracealyzer*** para PC. Existen dos maneras de grabar la trazabilidad:

- Modo Snapshot: funciona manteniendo los datos de la traza en un espacio de la memoria RAM, lo que permite capturar la traza de manera "instantánea" en cualquier momento y guardando el contenido de dicho espacio de la memoria RAM. Este modo está optimizado para la eficiencia de la memoria por lo que la tasa de transmisión de datos es de 10-20 KB/s dependiendo del sistema. Este modo se puede emplear en cualquier sistema, y en ocasiones se utiliza como caja negra durante pruebas de campo.
- Modo Streaming: los datos de la traza se envían continuamente al PC host permitiendo realizar un seguimiento de la traza durante periodos de larga duración. Para trabajar en este modo se puede hacer uso de sondas de depuración como:
    - *IAR I-Jet*
    - *Keil ULINKpro*
    - *SEGGER J-Link*
    - también se puede utilizar otras interfaces como *USB, TCP/IP, SPI, UARTs de alta velocidad o sistemas de ficheros.*

La librería de grabación proporcionada por ***Tracealyzer*** está en lenguaje C, y ambos modos de trabajo mencionados anteriormente comparten la API, por lo que es fácil cambiar el modo de trabajo de uno a otro.

Existen varias formas de conectar la tarjeta del sistema embebido con el PC haciendo uso de la librería *Trace Recorder* y son las siguientes:

- SEGGER RTT: Segger J-Link usa una función propia llamada Real-Time Transfer, RTT que consiste en buffers de RAM en la tarjeta del sistema las que el J-Link lee sin crear conflicto con la ejecución del sistema. ***Tracealyzer*** ocupa dos buffers, uno como canal de control de comandos de recepción y otro como canal de transmisión de datos de la traza. Existen placas que poseen J-Link on board, es decir, incorporados en la propia placa físicamente, por lo que no es necesario emplear uno externo. La desventaja es que estos suelen transferir una tasa de datos menor que uno propio.
- TCP/IP: Para transmitir los datos de la traza sobre una conexión en red, por ejemplo, una transferencia TCP/IP, seleccionando el puero TCPIP. El puerto TCP/IP actúa como servidor mientras que la aplicación ***Tracealyzer*** se conecta como cliente.
- Puerto Serie: Se puede transmitir la traza a través de un puerto de comunicaciones, COMx, lo que permite emplear una comunicación estándar basada en UART. Algunas tarjetas como las de la familia STM32 poseen drivers USB que facilitan esta comunicación para conectarlas físicamente con un cable USB al PC. Dichos drivers se trabajan con el entorno STM32Cube.
- Transmisión a través del puerto ITM de ARM: Interfaz para procesadores de ARM disponible en la familia de microcontroladores Cortex-M3/M4/M7, pero no disponible en los Cortex-M0/M0+.

De los dos modos de grabar la trazabilidad, en el proyecto se hace uso del **modo Streaming**, y para la conexión entre el microprocesador y el PC se hace uso del **puerto ITM** al que se conecta físicamente el depurador ***ULINKpro***.

La librería ***Trace Recorder*** es un software que está en código C y que se debe añadir al proyecto, pero no se debe añadir todos los ficheros, sino los esenciales ya que en este proyecto se trabaja con el puerto de **transmisión ITM**. Para ello se deben añadir los siguientes ficheros:

Tabla 5. Ficheros esenciales junto con su directorio de FreeRTOS

| | | |
|---|---|---|
| /config | **trcConfig.h** | La configuración principal de la librería. En este fichero se debe seleccionar el **modo Streaming** como en la figura 28. |
| /config | **trcSnapshotConfig.h** | Configuración específica del **modo Snapshot**. |
| /config | **trcStreamingConfig.h** | Configuración específica del **modo Streaming**. |
| /include | **trcRecorder.h** | La API pública de la librería. Es necesario incluirla en el fichero **FreeRTOSConfig.h** |
| /include | **trcHardwarePort.h** | Contiene todas las dependencias hardware, puertos hardware predefinidos incluido el propio para **Cortex-M** de **ARM**. |
| /include | **trcKernelPort.h** | Definiciones específicas para **FreeRTOS**. |
| /include | **trcPortDefines.h** | Constantes para los archivos de configuración. |
| /streamports/ARM_ITM<br>/streamports/ARM_ITM<br>/streamports/ARM_ITM | **trcStreamingPort.c**<br>**trcStreamingPort.h**<br>**Keil-uVision-Tracealyzer-ITM-Exporter.ini** | Fichero principal y de configuración del **puerto ITM**.<br><br>Software que se encarga de exportar la trazabilidad a un archivo llamado ***Tracealyzer.psf*** ubicado en el PC. |
| / | **trcSnapshotRecorder.c** | Implementación del **modo Snapshot**. |
| / | **trcStreamingRecorder.c** | Implementación del **modo Streaming**. |
| / | **trcKernelPort.c** | Definiciones y funciones específicas de **FreeRTOS**. En especial crea y configura la tarea ***TzCtrl.*** |

La librería ***Trace Recorder*** usa un buffer interno en memoria **RAM** en forma de páginas donde los eventos son escritos y transferidos al PC periódicamente a través de la tarea ***TzCtrl.*** Se envían las páginas que están llenas, pero no las que se encuentren recibiendo eventos.

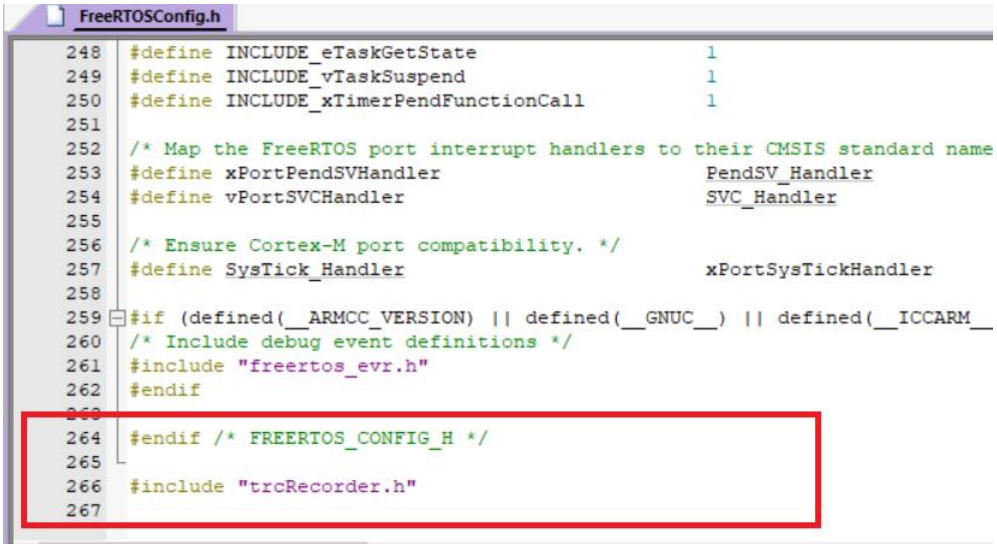



Figura 26. Inclusión del fichero *trcRecorder.h* mediante una directiva en el fichero *FreeRTOSConfig.h*

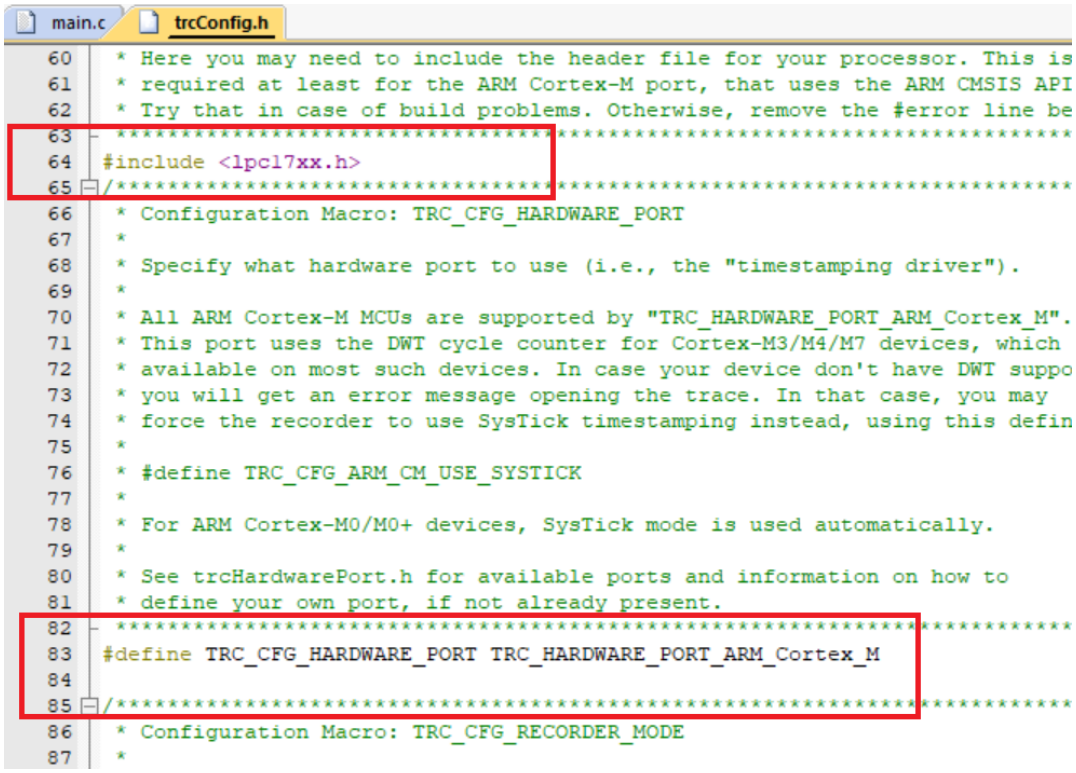



Figura 27. Inclusión del fichero del procesador LPC1768 y definición del puerto para un procesador Cortex-M

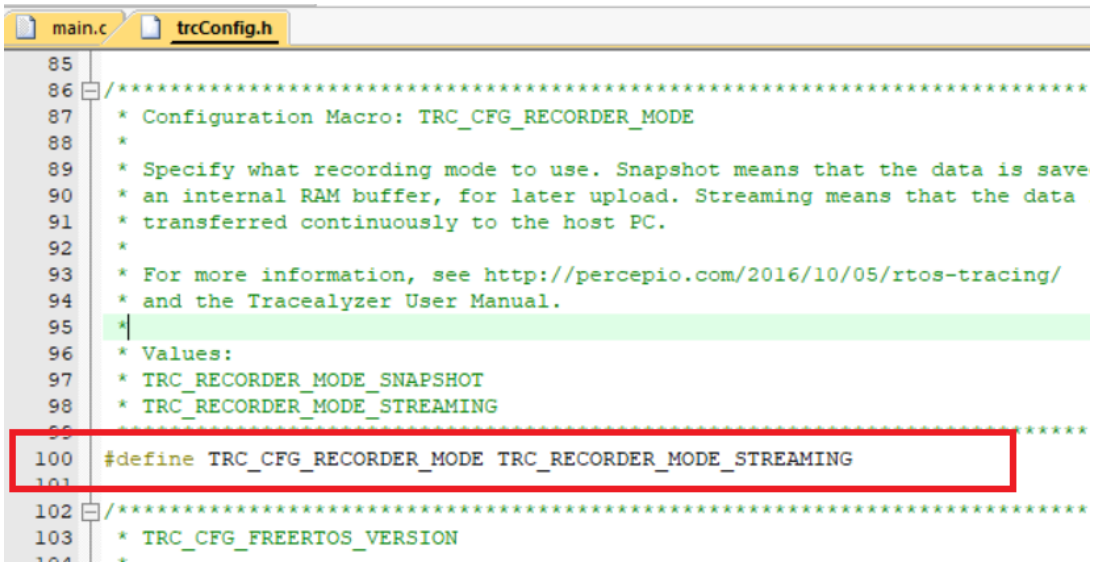



Figura 28. Configuración en modo Streaming

Para poder incluir el fichero de inicialización se debe abrir la ventana ***Options Target*** y seleccionar dentro de la opción ***DEBUG*** la sección **Fichero de Inicialización** e incluir el archivo ***Keil-uVision-Tracealyzer-ITM-Exporter.ini***. Ese archivo exporta la trazabilidad a un fichero localizado en el PC y facilita la opción de iniciar y parar de grabar la trazabilidad con dos botones de manera visual.

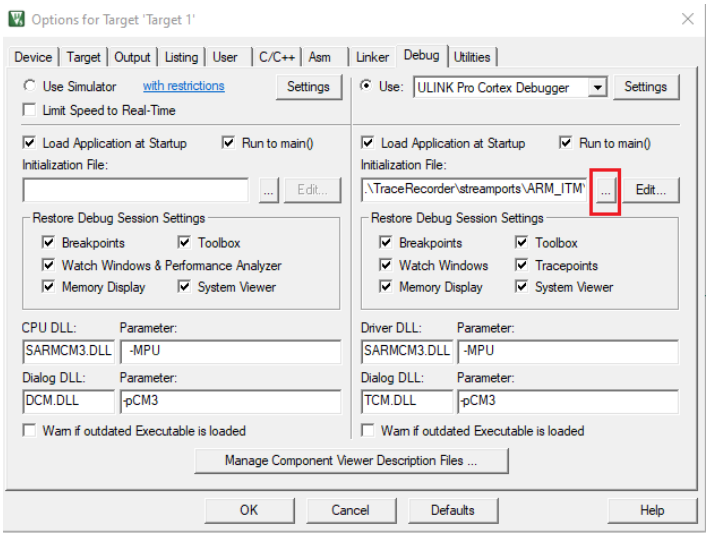



Figura 29.a. Ventana de opciones de tarjeta de KEIL

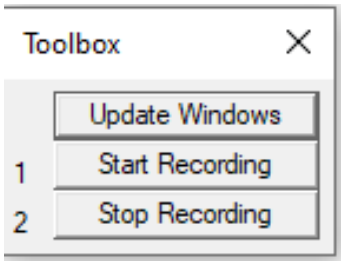



Figura 29.b. Botones para iniciar y parar la trazabilidad

En la ventana ***Options Target*** en la opción ***Debug***: *Use* se debe seleccionar el depurador ***ULINKpro Cortex Debugger*** porque es el que se va a emplear en el proyecto, y dentro de su configuración en ***Settings*** aparece la ventana ***Traget Driver Setup*** donde se debe seleccionar el **puerto *SW*** y la velocidad de trabajo que se desee. En la **pestaña *Trace*** se debe habilitar la traza y verificar que se tienen habilitados los **puertos ITM**. En la siguiente figura se muestra cómo debería estar hecha la configuración.

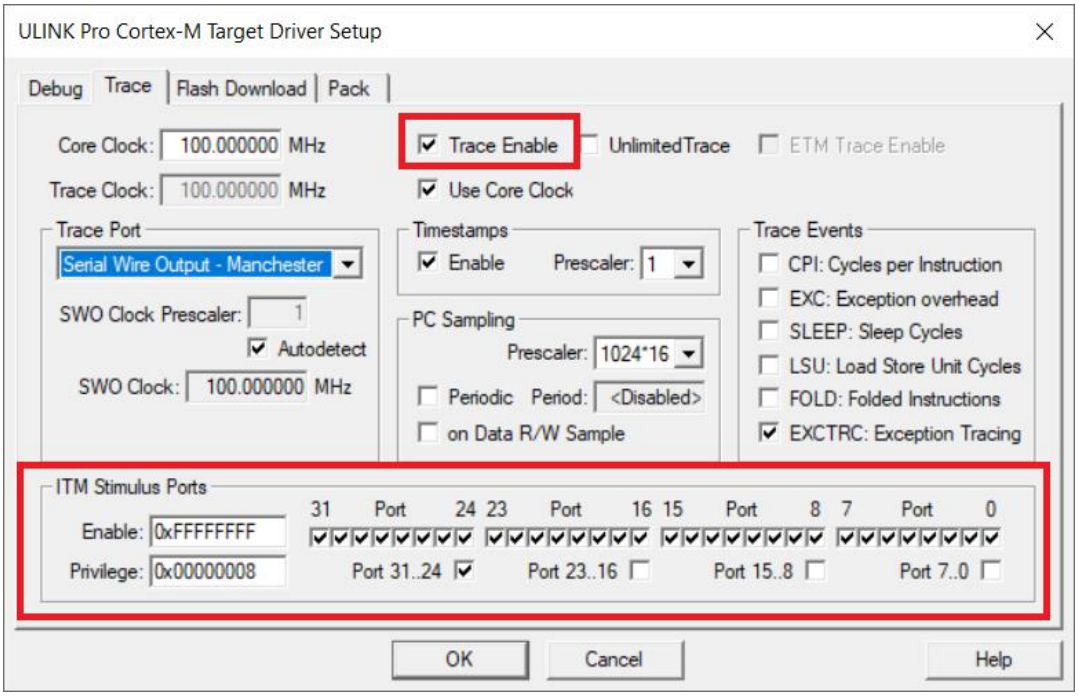



Figura 30. Ventana de configuración sección traza

## 7.3. Modo Streaming sobre la interfaz ITM usando ULINKpro

El **modo Streaming** sobre la **interfaz ITM** de ARM está disponible para microcontroladores de ARM basados en **Cortex-M3, M4 y M7**. Esta interfaz es compatible con:

- *FreeRTOS, Micrium uC/OS-III y* disponible en un futuro para *SafeRTOS* y *ThreadX*.

La ventaja de esta interfaz es que se pueden transmitir los datos sobre el pin SWO disponible en todos los conectores para depuración de todos los Córtex de ARM.

No es necesario escribir en la memoria RAM del procesador a depurar, porque la **interfaz ITM** escribe los datos directamente en sus propios registros, escribiendo a una velocidad de una palabra de 32 bits en menos de 10 ciclos de reloj. Se recuerda que el **módulo ITM** es un periférico físico integrado en el microcontrolador. Típicamente suele ser un buffer que almacena 10 registros y los datos de la traza se generan en ráfagas que rellenan el buffer. Esta forma de rellenar el buffer es muy importante a la hora de trabajar a una alta velocidad sobre el **pin SWO**, que normalmente lo hace a 30 MHz en codificación Manchester, para que el depurador pueda enviar los datos al PC lo suficientemente rápido. En ocasiones este puerto se bloquea cuando el buffer propio del **ITM** está lleno, y el peor caso sucede cuando se necesita transmitir 10 bytes del buffer ITM sobre el **pin SWO**. Experimentalmente, si se trabaja a 24 MHz sobre el **pin SWO**, se tiene un bloqueo de 15 us. Si se trabaja con un depurador rápido como el ***ULINKpro*** este bloqueo no suele producirse.

***Percepio Tracealyzer*** provee un software llamado ***Keil-uVision-Tracealyzer-ITM-Exporter.ini*** mencionado en el apartado anterior, que exporta toda la trazabilidad a través del puerto 1 al fichero ***tracealyzer.psf*** situado en el espacio de trabajo del proyecto creado en el PC.

## 7.4. Configuración de Tracealyzer

Para que la herramienta ***Tracealyzer*** procese el fichero ***Tracealyzer.psf*** generado en **modo *Streaming***, se debe configurar dentro de su ventana de opciones, la sección ***PSF Streaming Settings*** donde se indica el tipo de conexión con la tarjeta y la localización del fichero ***Tracealyzer.psf***

El fichero ***Traealyzer.psf*** se genera cuando se entra en modo depuración con ***KEIL uVision5*** y se debe abrir con ***Percepio Tracealyzer*** para observar la trazabilidad de manera gráfica.En el siguiente apartado se explicará la terminología de la herramienta de depuración ***Percepio Tracealyzer***.

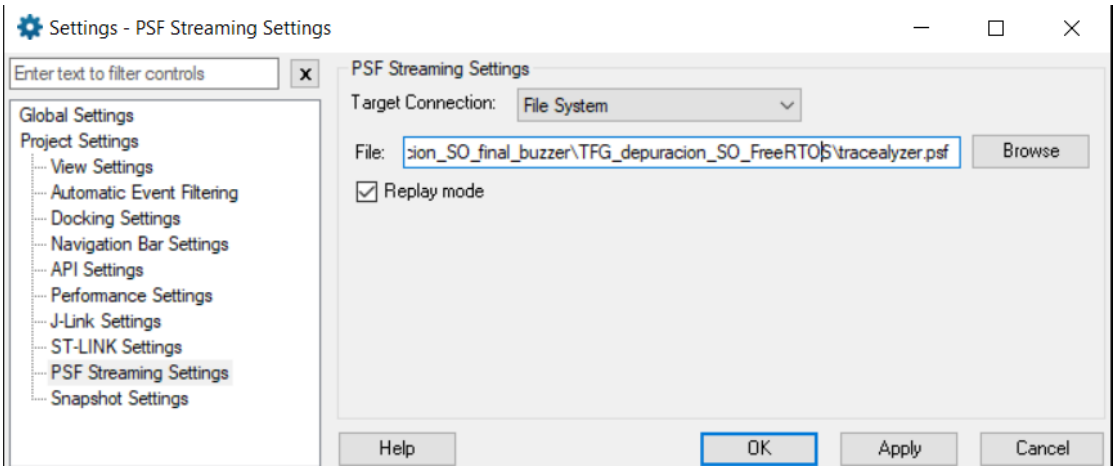



Figura 31. Configuración del modo PSF Streaming

## 7.5. Terminología de Tracealyzer

Tal y como se ha mencionado en el apartado anterior denominado ***Introducción a Percepio Tracealyzer,*** la herramienta de depuración ofrece más de 30 formas de observar la trazabilidad en forma de ventanas con elementos gráficos, listas e incluso máquinas de estados. Pero para poder trabajar con la herramienta es necesario aclarar el significado de la terminología que emplea ***Tracealyzer***. La siguiente lista lo detalla brevemente.

- ***Tarea:*** Se entiende como un hilo de ejecución. ***Tracealyzer*** se refiere de manera general a los hilos como "tareas".
- ***Actor:*** Puede ser una tarea, un hilo o una rutina de interrupción/ISR.
- ***Actor Instance:*** Iteración completa de un actor. Para las ISRs significa ejecutar por completo la ISR desde que se inicia hasta que finaliza. Para tareas, significa la iteración completa de una tarea. Se entiende que cada tarea tiene un bucle principal y una iteración de la terea consiste en completar dicho bucle principal una vez.
- ***Execution Time:*** tiempo de ejecución, cantidad de tiempo que emplea la CPU en ejecutar una instancia de actor. Si otra tarea de mayor prioridad interrumpe a la tarea en ejecución, el tiempo en ejecutar la nueva tarea no es incluido.
- ***Response Time:*** tiempo desde el inicio de una Instancia de Actor hasta que es finalizado. Se incluye el tiempo que gasta la CPU en ejecutar otras tareas que interrumpan. Si los eventos "Actor Ready" están incluidos en la traza, cuentan como el punto de inicio del tiempo de respuesta.
- ***Fragment:*** un fragmento es un intervalo durante el que se ejecuta un actor sin ser interrumpido. Cada fragmento se visualiza como un rectángulo de color sólido en la ventana *Trace View*. Un fragmento pertenece específicamente a una instancia de actor propia.
- ***Fragmentation:*** significa el número de fragmentos dentro de un *Actor Instance*. Si un *Actor Instance* se ejecuta en un tiempo completo sin ser interrumida o fragmentada por otra tarea, la fragmentación es 1.
- ***Object:*** un objeto se entiende como una cola, semáforo, mutex o tarea en el contexto de referirse a algo en un evento de la traza.
- ***Service:*** Típicamente una función de la API proporcionada por el kernel, realizando una operación en un objeto.
- ***Periodicity:*** Tiempo entre dos instancias de actor consecutivas, válidas desde el inicio de la instancia del actor anterior hasta el inicio de la instancia del actor actual. Siempre referente la misma tarea.
- ***Separation:*** Tiempo entre el fin de la instancia del actor previo y el inicio de la instancia de actor actual, para una misma tarea.

# 8. Recursos de Tracealyzer

En este apartado se detallarán varios recursos que proporciona ***Tracealyzer*** demostrando sus ventajas a la hora de depurar el S.O. ***FreeRTOS*** ejecutándose en la placa Mini-DK2. Para demostrar los distintos recursos se analiza el sistema expuesto en apartados anteriores. Se recuerda que el sistema trabaja con dos tareas.

## 8.1. Live Stream

Mientras se está grabando una traza, se puede cuantificar la utilización de la CPU del sistema embebido gracias a la ventana **LIVE STREAM**, donde se puede observar la cantidad de **bytes/s** de traza recibidos, el número de eventos totales recibidos por segundo y el número de eventos perdidos. Esto puede resultar muy útil si hubiera algún tipo de problema en la conexión con la tarjeta. Si se desea conocer de manera rápida si el sistema está sufriendo algún tipo de bloqueo, basta con observar la gráfica y buscar que la carga de la CPU sea del 100% cuando no debería serlo. Se muestra la carga de la CPU en tanto por ciento representando el tiempo que se gasta ejecutando una tarea, en relación con el tiempo total de procesamiento. Para entender esto se expone un ejemplo sencillo, si una tarea se ejecuta durante 10 segundos, y durante los diez segundos no realiza ningún retardo empleando la función ***vTaskDelay()***, esa tarea consume un 100% de la carga de la CPU. En cambio, si emplea algún tipo de retardo propio de ***FreeRTOS*** , como el retardo ***vTaskDelay(),*** durante 5 segundos, esa tarea consume un 50%. De manera general cuando una tarea llama a algún tipo de retardo del sistema operativo, como el retardo ***vTaskDelay(),*** la CPU permanece sin ejecutar ningún proceso.

En la siguiente figura se muestra la carga de la CPU de la tarjeta del sistema, que tiene un consumo constante del 100%. A simple vista uno puede pensar que se tiene algún tipo de bloqueo, pero la realidad es que la tarea ***Calcula_Pulsos y Calcula_Oxígeno*** consumen el 100% de la CPU, por lo que el consumo corresponde con el 100% como se ilustra en la siguiente figura.

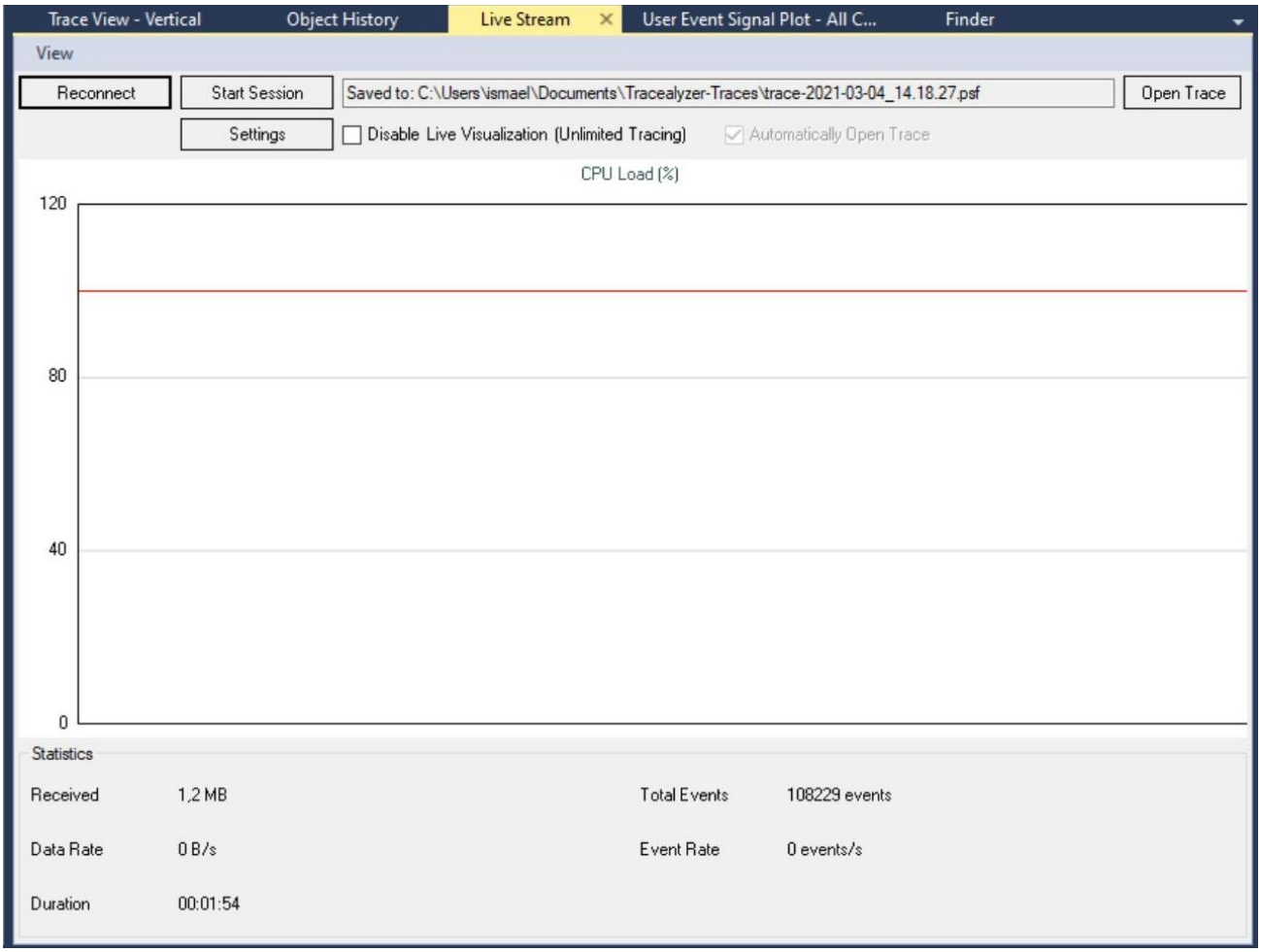



Figura 32. Visualización de la carga de trabajo en la CPU del sistema

## 8.2. Eventos de usuario – Canales

Se pueden crear logs de cualquier evento o de cualquier dato de la aplicación haciendo uso de las funciones de la librería de grabación ***Trace Recoder library*** de tal manera que estos logs creados pasan a llamarse ***User Events*** o eventos de usuario, y que pueden visualizarse en distintas ventanas de ***Tracealyzer*** como ***Trace View, Event log*** y ***User Events***. Si se crean logs de transiciones entre estados en el sistema como eventos de usuario, se pueden crear máquinas de estados basadas en esos eventos.

En el programa principal del software, ***main.c***, se han creado cuatro canales de depuración visibles a través de ***Tracealyzer*** con el nombre de ***String Channel***, $SpO_2$ , ***BPM*** y ***Clock***.

Dichos canales consisten en cuatro registros que almacenan información en forma de cadena da caracteres, como el valor del nivel de oxígeno en sangre en %, el valor de pulsos por minuto y la fecha y hora. Esta información es muy útil porque se puede visualizar el contenido de dichos registros en cuando se esté depurando el sistema con ***Tracealyzer.***

También se pueden generar gráficas con los valores numéricos en dichos canales a lo largo del tiempo, sin necesidad de ver la información en la pantalla o sin tener que conectarse al puerto serie. En apartados posteriores se empelan estos canales para visualizar distinta información de la trazabilidad y ***Tracealyzer resaltará estos canales en color amarillo.***

## 8.3. CPU Load Graph

La gráfica de consumo de CPU ilustra el uso que hacen los **actores** o tareas a lo largo del tiempo. Por defecto, muestra todos los actores. Divide la traza en un cierto número de intervalos en forma de barras. Para indicar cantidades de tiempo, ***Tracealyzer*** emplea la siguiente nomenclatura:

- *( **m : s . ms . us** )* siendo ***m*** minutos, ***s*** segundos, ***ms*** milisegundos y ***us*** microsegundos.

Cada rectángulo dentro de una barra representa el consumo de un actor, por lo que si en un intervalo de tiempo hay varios actores consumiendo la CPU se observarán distintos colores para cada uno de los actores.

Tabla 6. Detalle de las Tareas del sistema

| Tarea | Nombre | prioridad | Color en Tracealyzer | Utilización total de CPU *(%)* |
|---|---|---|---|---|
| Tarea 1 | Calcula_Oxigeno | 2 | Verde | 33.8 |
| Tarea 2 | Calcula_Pulsos | 3 | Amarillo | 47.55 |
| Tarea 3 | Startup | 2 | Azul | 0.25 |
| Tarea 4 | Tmr Svc | 3 | Rojo | 0.00 |

En la siguiente figura se muestra el resultado del análisis de la traza capturada. A primera vista, las dos tareas ***Calcula_Oxígeno y Calcula_Pulsos*** de color verde y amarrillo respectivamente, se ejecutan durante más tiempo que el resto de las tareas. Se recuerda que una instancia es una iteración completa de una tarea de su bucle principal correspondiente.

Esta herramienta es útil si se quiere analizar de manera general la duración de las tareas intentando buscar algún error de manera rápida. Por ejemplo, analicemos la duración de la tarea **Tmr Svc**, que se representa en color rojo, en con este recurso. Se recuerda que la tarea **Tmr Svc** está suspendida inmediatamente después de ser creada, por lo que lo esperado sería que no se ejecute en ningún momento. Con la ventana **CPU Load Graphs** se visualiza de manera muy rápida que la tarea **Tmr Svc** no se ha ejecutado en ningún momento, pues no se aprecia ninguna barra de color rojo.

Por otro lado, se representa de color azul la transición de inicializar el sistema operativo y se nombra como ***startup***. Esta transición solo debe ocurrir en el inicio y en un breve periodo de tiempo, por lo que esta ventana permite visualizar una correcta inicialización del sistema. En la siguiente figura el resultado es el esperado tanto para la tarea **Tmr Svc** como para **startup**.

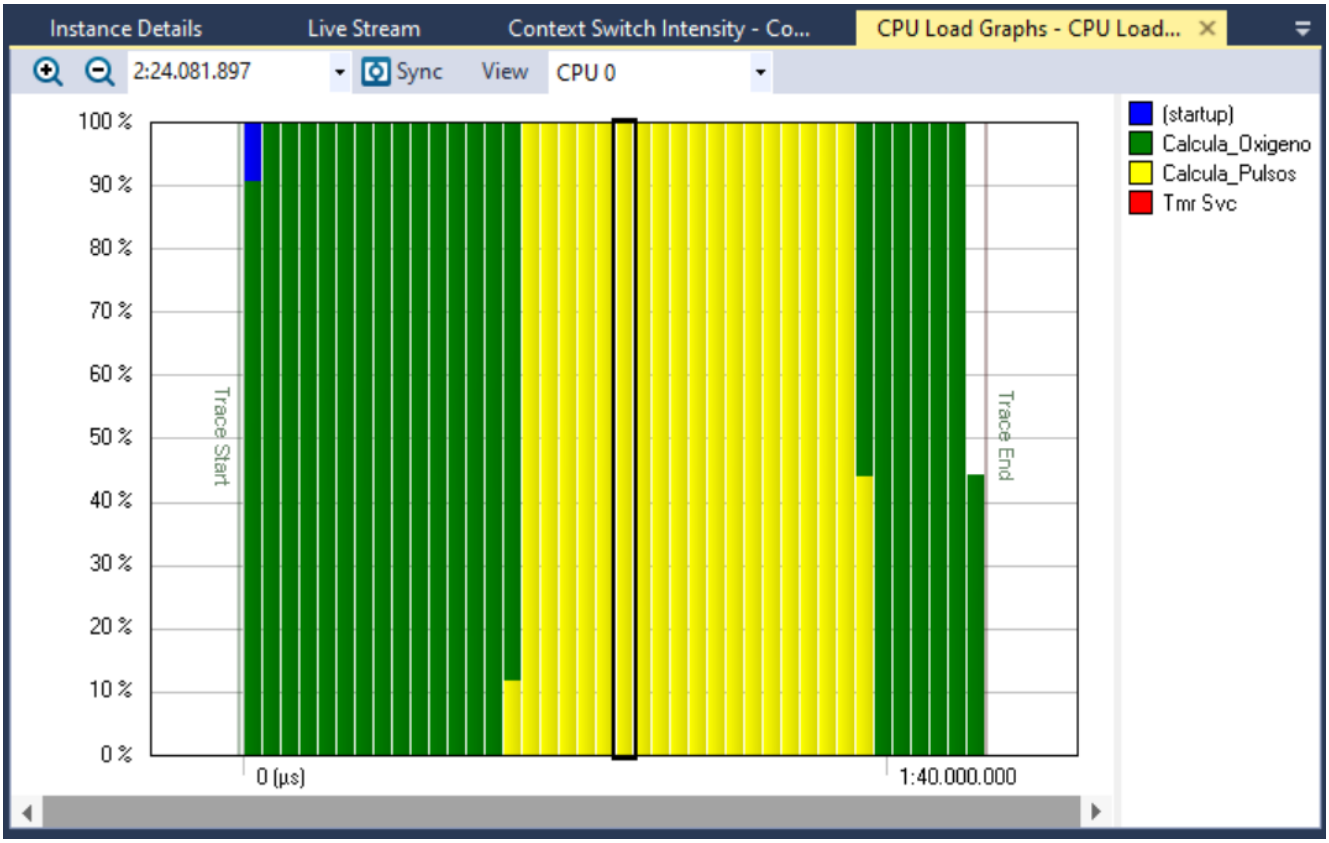



Figura 33. Resultado del consumo de la CPU durante la grabación de la trazabilidad

Se puede hacer zoom en los intervalos que se desee para ver en detalle los actores que se están ejecutando. De manera automática la aplicación dividirá el tiempo seleccionado para ampliar en varias barras de duración más reducida. Por ejemplo, se ha hecho zoom en la parte del inicio. El resultado es que la tarea ***Startup*** se ejecuta durante alrededor de 200 ms, y que a continuación se ejecuta la tarea ***Calcula_Oxígeno***.

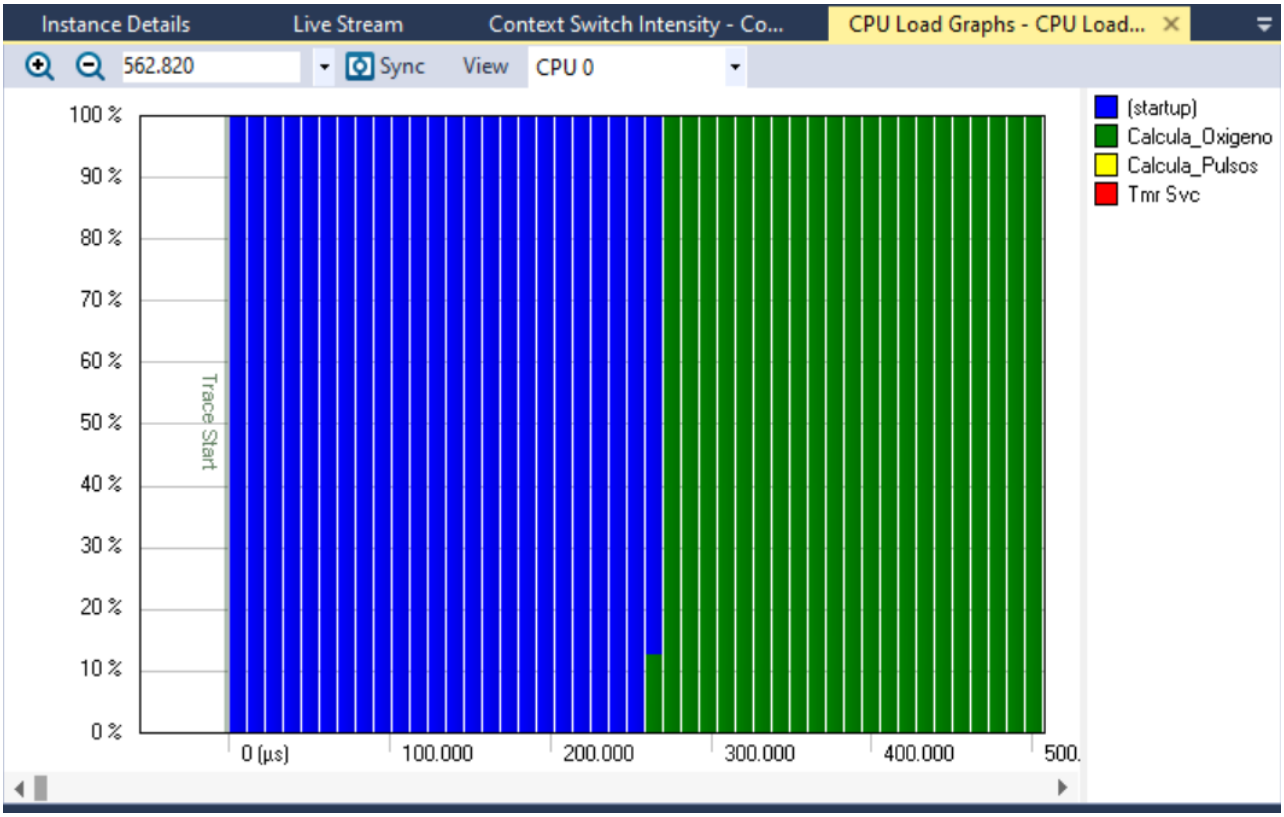



Figura 34. Zoom en el inicio para ver en detalle la duración de la tarea Startup

Seleccionando cada tarea desde esta ventana, se muestra una ventana llamada ***Selection Details.*** Gracias al recurso ***Selection Details*** se ha creado la siguiente tabla donde se indica la utilización de cada tarea en la CPU durante toda la trazabilidad grabada.

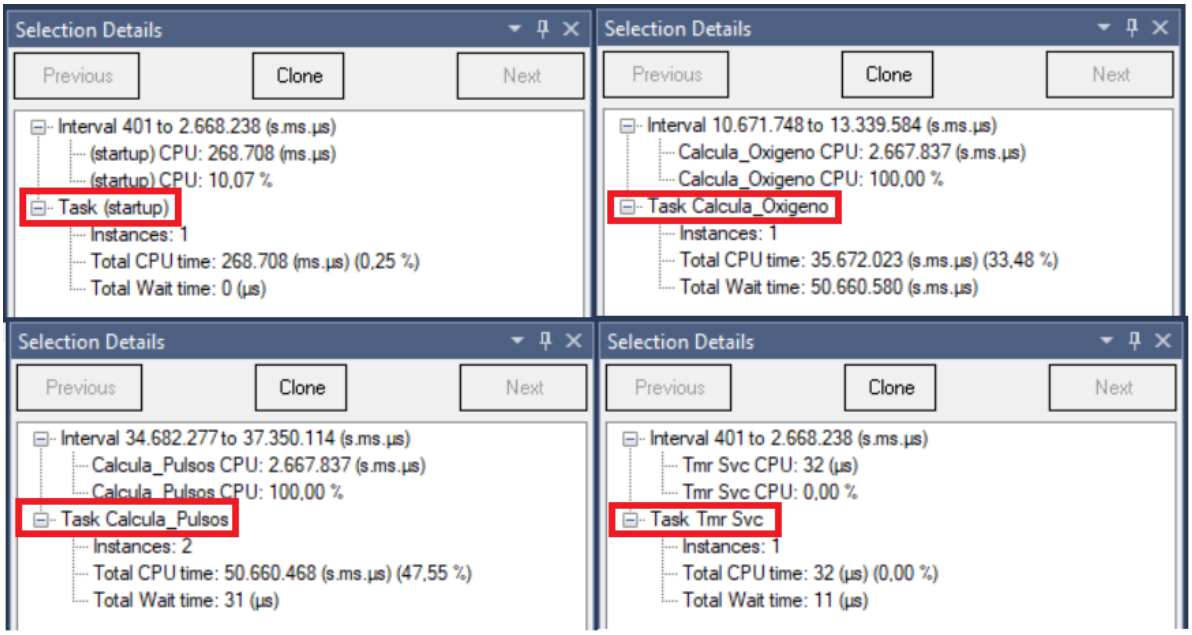



Figura 35. Detalles de la tarea 1, 2, 3 y startup

Tabla 7. Consumos de las tareas del sistema representados en porcentaje y en segundos

| | Startup | Calcula_Oxígeno | Calcula_Pulsos | Tmr Svc |
|---|---|---|---|---|
| Uso total de CPU *(%)* | 0.25 | 33.48 | 47.55 | 0.19 |
| Tiempo total en estado de ejecución *(m:s.ms.us)* | 268.708 | 35.672.023 | 50.660.468 | 32 |
| Tiempo total en estado de espera *(m:s.ms.us)* | 0 | 50.660.580 | 31 | 10 |
| Instancias | 1 | 1 | 1 | 1 |

***Tracealyzer*** tiene un pequeño fallo, un ***bug,*** y consiste en que la última tarea que se ejecuta durante la grabación de la traza, aunque se muestre visualmente en la ventana ***Trace View***, no se registran sus datos y es como si no existiera. Por ejemplo, durante esta traza capturada, debería contarse con una instancia adicional de la tarea ***Calcula_Pulsos*** como se ve en la siguiente figura, sin embargo, esta última instancia de la tarea ***Calcula_Pulsos*** no se tiene en cuenta, es por eso que la tarea ***Calcula_Puslsos,*** en la figura 33, no aparece en la parte final ***Trace End*** cuando en realidad debería aparecer***.*** Por ese motivo, si se suman los porcentajes de la tabla anterior, falta un 18% que corresponde con esta instancia no registrada por el ***bug.***

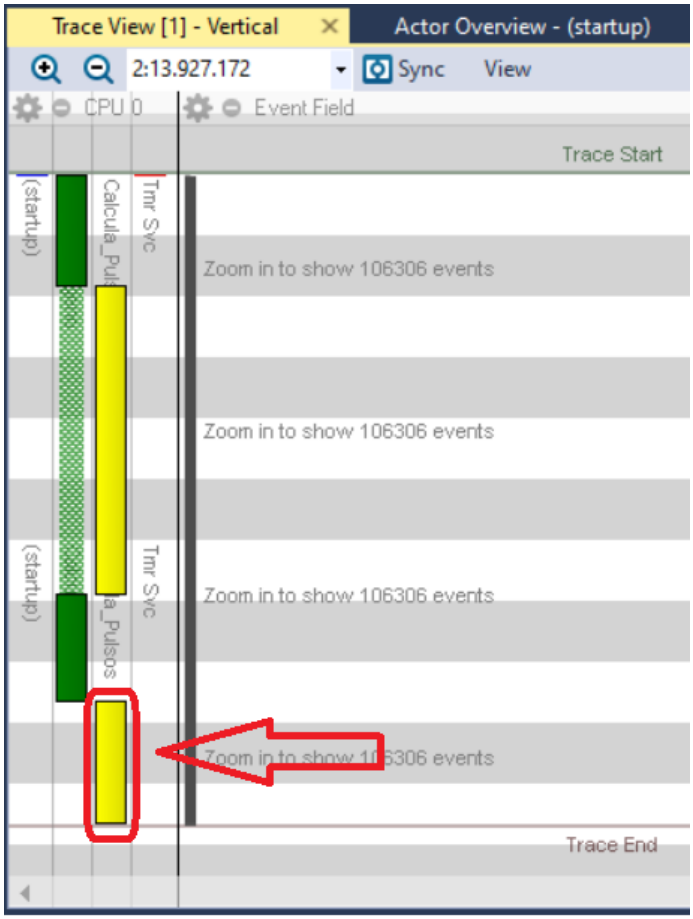



Figura 36. Ventana **Trace View** de las tareas capturadas en la grabación de la traza

Tal y como se ha mencionado en el apartado ***Introducción a Percepio Tracealyzer,*** se pueden enlazar unos recursos con otros haciendo doble clic o buscando alguna opción clicando con el botón derecho sobre algún dato. En este ejemplo se ha hecho clic con el botón derecho sobre el rectángulo de la tarea ***Calcula_Oxígeno*** y se ha abierto la opción ***Actor Overview*** donde se pueden ver todas las instancias de un actor en una lista.

El resultado queda reflejado en la figura 38. En la ventana ***Actor Overview*** aparece el inicio y fin de la única instancia de la tarea ***Calcula_Oxígeno***, el tiempo de ejecución y el tiempo de respuesta. También se indica el número de fragmentos que tiene la instancia.

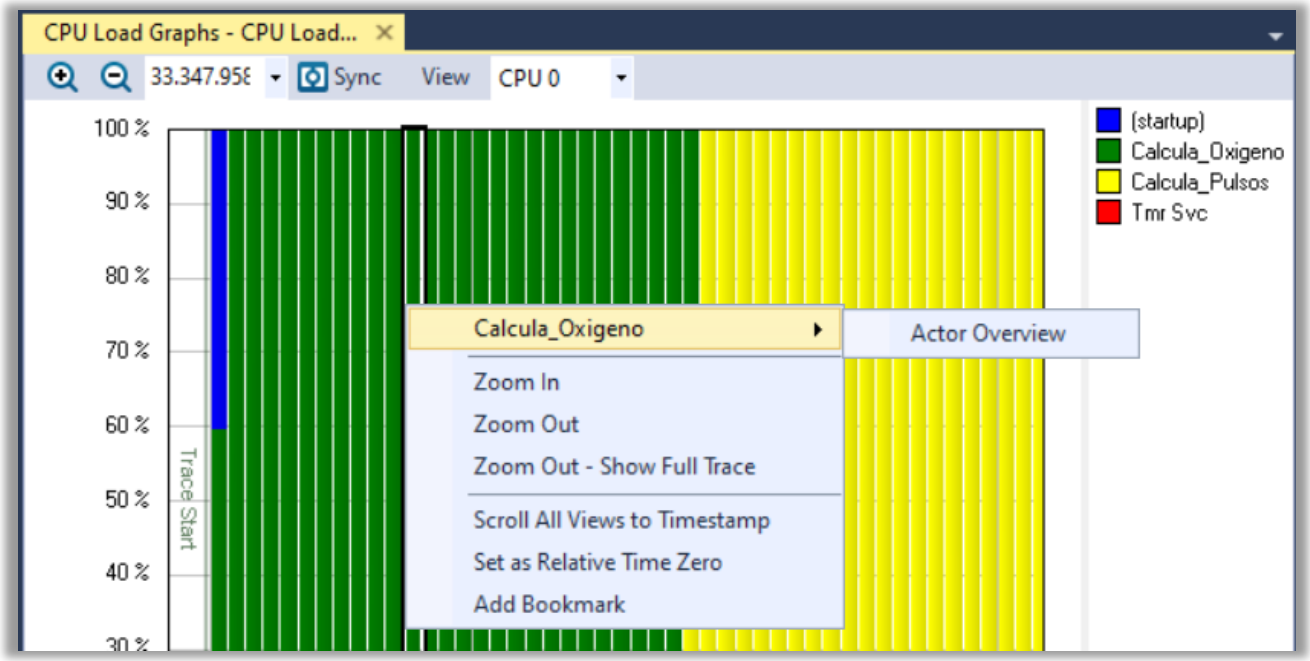



Figura 37. Selección de la ventana *Actor Overview* desde la ventana *CPU Load Graphs*

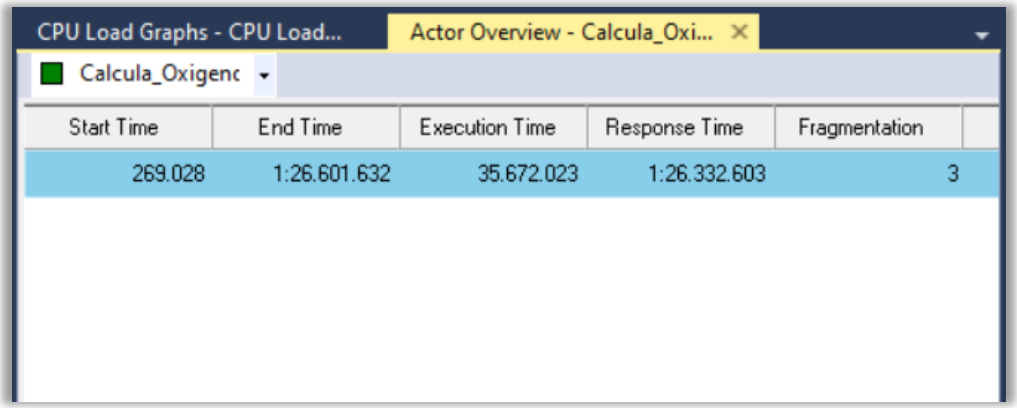



Figura 38. Datos proporcionados por la ventana *Actor Overview*

Haciendo doble clic en la instancia de la anterior figura, aparece una ventana llamada ***Selection Details*** donde se muestran los siguientes datos:

- La tarea consta de una sola instancia
- La instancia se ha fragmentado en 3 partes
- La instancia seleccionada ha sido lanzada por la tarea *startup*
- la tarea ***Calcula_Oxígeno*** lanza a otras tareas en 3 ocasiones
- La tarea usa un 33.5% de la CPU
- El nivel de prioridad de la tarea es ***2***

A la derecha en la sección ***Selection Details*** haciendo clic en el símbolo "+" en la parte de *Triggers* se puede ver que la ***Calcula_Oxígeno*** ha lanzado a la tarea ***Tmr Svc*** en el tiempo 269.184 (*ms.us*) y a la ***Calcula_Pulsos*** en dos ocasiones.

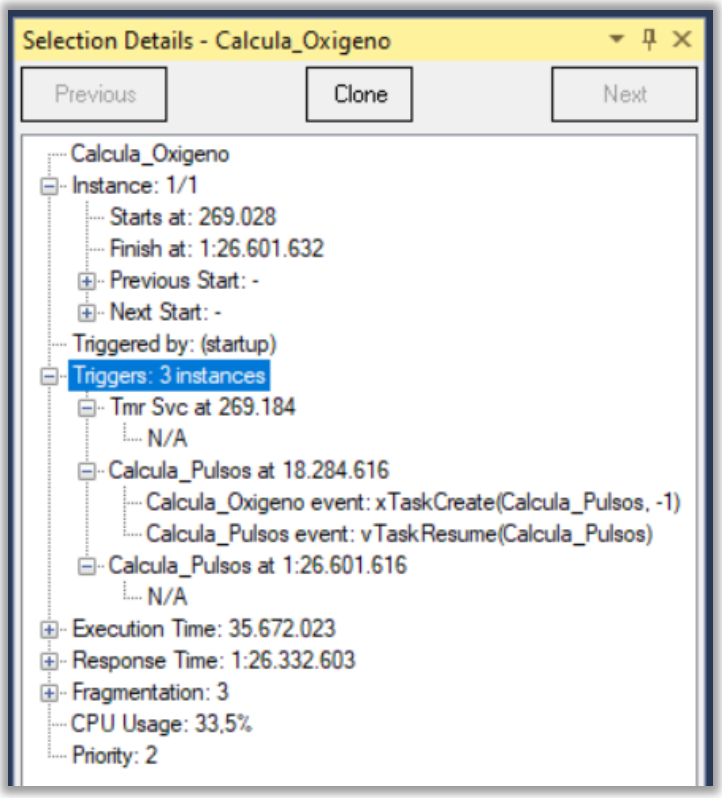



Figura 39. Instance Details

Para ver en detalle lo que ha ocurrido se hace doble clic en **Tmr Svc**, de manera que se puede visualizar desde la ventana ***Trace View*** el momento en que la tarea ***Calcula_Oxígeno*** lanza a la tarea ***Tmr Svc.***

En esta ventana se observa claramente que la tarea ***startup*** (en azul) finaliza lanzando a la tarea ***Calcula_Oxígeno*** (en verde), y que la tarea ***Calcula_Oxígeno*** (en verde) lanza a la tarea ***Tmr Svc*** (en rojo) pero esta es inmediatamente suspendida, volviendo el CPU a ejecutar la tarea ***Calcula_Oxígeno***. De este modo se puede analizar cualquier evento del sistema de manera gráfica fácil e intuitivamente y observar eventos haciendo doble clic redirigiéndose a la ventana ***Trace View,*** ventana que se detallará en capítulos siguientes.

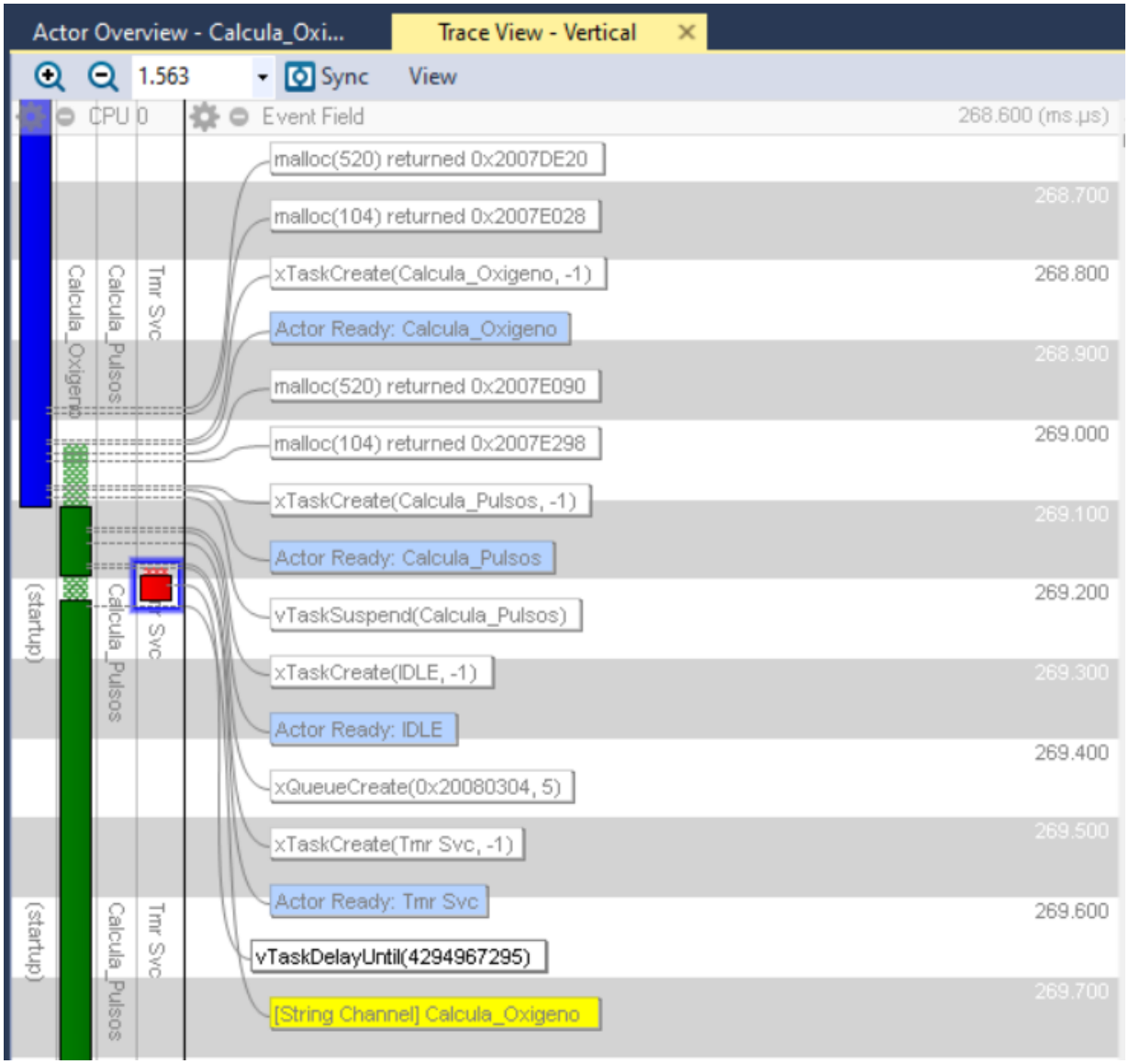



Figura 40. Trace View - Vertical

## 8.4. Event Log

Este recurso permite visualizar la trazabilidad en forma de una lista o listado y además tiene una opción que permite filtrar los resultados de una búsqueda. También permite exportar la trazabilidad como un fichero de texto o en formato **CSV**. Haciendo doble clic en cualquier evento hace que se muestre el evento en la ventana ***Trace View,*** ventana que se explicará en el siguiente apartado. Resulta muy útil a la hora de analizar cómo se ha inicializado el sistema.

En la siguiente figura se muestra que la primera tarea en ejecutarse es la tarea ***startup*** donde se ha recibido a través de ***String Channel*** el mensaje ***Creating Tasks*** en el tiempo 456 us.

Posteriormente se crea la tarea ***Calcula_Oxígeno*** reservando en memoria 520 bytes, esto tiene sentido porque como se ha mencionado en el apartado anterior ***"6.5. Descripción de las tareas del sistema embebido"*** cada tarea requiere de 128 registros, de 4 bytes cada registro, por lo que ocuparía en total 512 bytes cada tarea,

además, cada tarea requiere un bloque de control de tarea (TCB) y un tamaño reservado en memoria **heap**, ocupando adicionalmente 104 bytes.

Una vez creada la tarea ***Calcula_Oxígeno***, adquiere el estado ***Ready***. Ocurre lo mismo para la tarea ***Calcula_Pulsos***. Después de la tarea ***startup*** se ejecuta la tarea ***Calcula_Oxígeno.*** Esta transición sucede en el tiempo 269.109 (*ms.us*).

En la ventana ***Event Log*** se respetan los colores asociados a cada tarea y se representan en un recuadro pequeño. Se indica el tiempo, color de la tarea, y la descripción del evento.

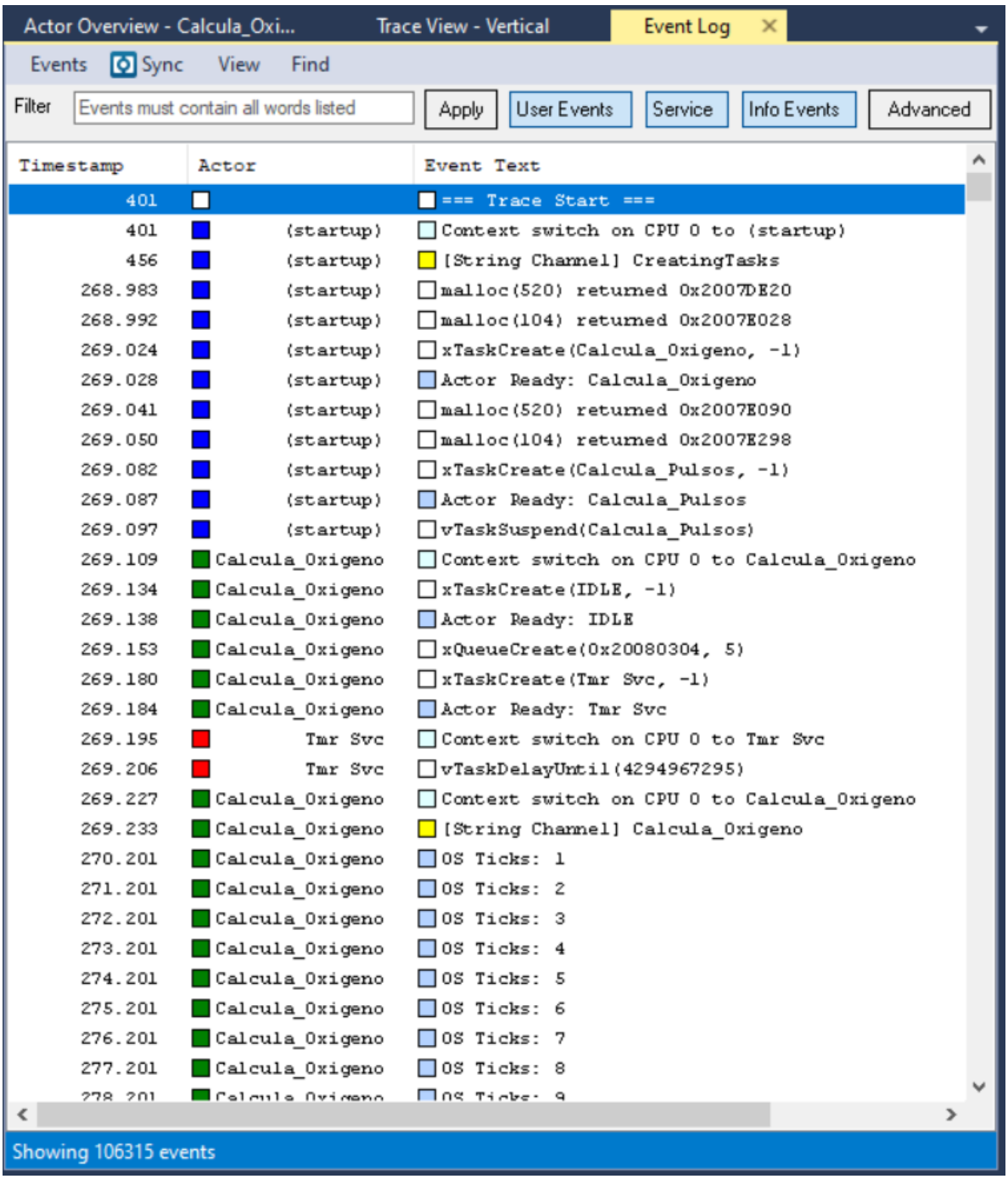



Figura 41. Datos proporcionados por la ventana Event Log

El propio sistema operativo posee un contador denominado ***OS_Tick*** que se incrementa cada milisegundo transcurrido, por lo que es muy común emplearlo como un reloj del sistema y para medir tiempos. Con el recurso ***Event Log*** se puede ver con exactitud en que instante se inicia el contador ***OS_Tick***, que suele ser en el inicio. En la siguiente figura se muestra que el contador ***OS_Tick*** se inicia dentro de la tarea ***Calcula_Oxígeno*** en el tiempo 270.201 (*ms.us*).

La ventana ***Event Log*** permite realizar una búsqueda de un evento en concreto, por ejemplo, cuando el sistema operativo emplea el canal ***Clock*** para enviar la fecha y hora actual hacia ***Tracealycer*** en depuración. Se recuerda que los canales creados por el usuario para transmitir datos se resaltan en amarillo. Por tanto, se debe escribir en el buscador "clock" tal y como se indica en la siguiente figura. Esto significa que cuando se capturaba la traza en depuración, el sistema ha mostrado la fecha y hora dos veces en los tiempos 10.305.861 (*s. ms.us*) y en 56.571.145 (*s. ms.us*) en la tarea ***Calcula_Oxígeno y Calcula_Pulsos*** respectivamente.

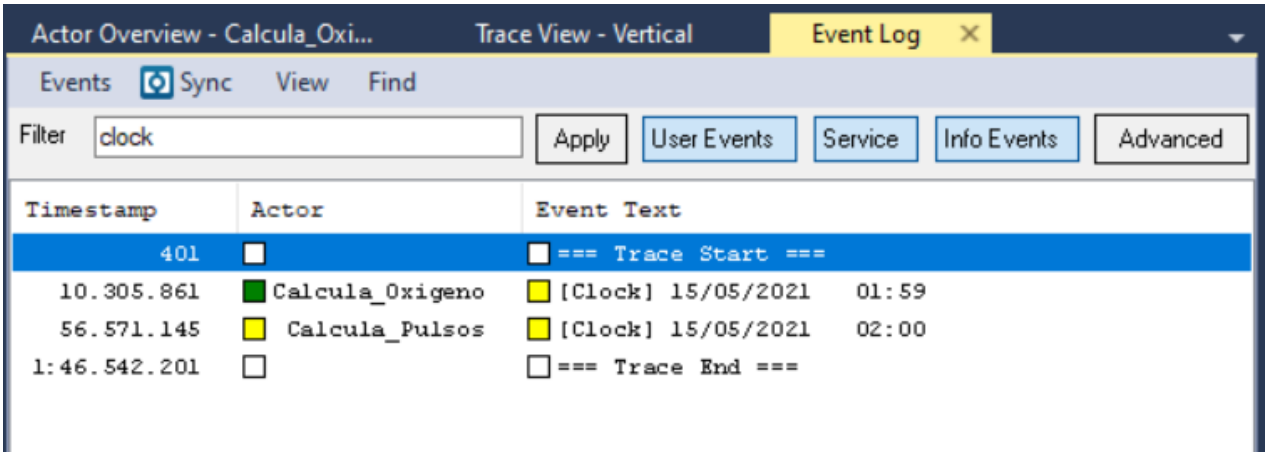



Figura 42. Búsqueda de la información enviada a través del canal clock

## 8.5. Trace View - Vertical

Esta ventana muestra todos los eventos grabados en una línea temporal en vertical. Opcionalmente se puede cambiar la vista a modo horizontal. En la figura 43 se ha seleccionado una línea temporal en vertical enfocándose en la tarea ***startup***, y en la figura 45 se ha seleccionado una línea temporal en horizontal mostrando toda la traza capturada, desde el inicio hasta el final. En ambas opciones se puede hacer **zoom** para ampliar la vista y visualizar más eventos si los hubiera. Si se hace doble clic en cualquier instancia de cualquier tarea se accede a la ventana ***Instance Details*** donde se muestran los detalles de la instancia seleccionada. Por ejemplo, se ha hecho doble clic en la primera instancia de la tarea ***Calucla_Oxígeno*** desde la ventana ***Trace View***, la aplicación ***Tracealyzer*** automáticamente nos dirige a la ventana ***Instance Details*** desde donde se puede analizar muchos más datos acerca de la instancia seleccionada. En el siguiente apartado se hablará sobre este recurso.

En la ventana ***Trace View*** se puede observar la ejecución de cada tarea a lo largo del tiempo, cada tarea se representa con su color descrito en apartados anteriores, y se divide la ventana en varias franjas con una marca temporal indicando el tiempo.

- Cuando una tarea es ejecutada por la CPU, se representa de un color sólido
- Cuando una tarea pasa a estar suspendida, su color desaparece
- Cuando una tarea es desalojada por otra de mayor prioridad, el color de la tarea desalojada es similar a una malla o red, y el color de la tarea de mayor prioridad pasa a ser de color sólido.

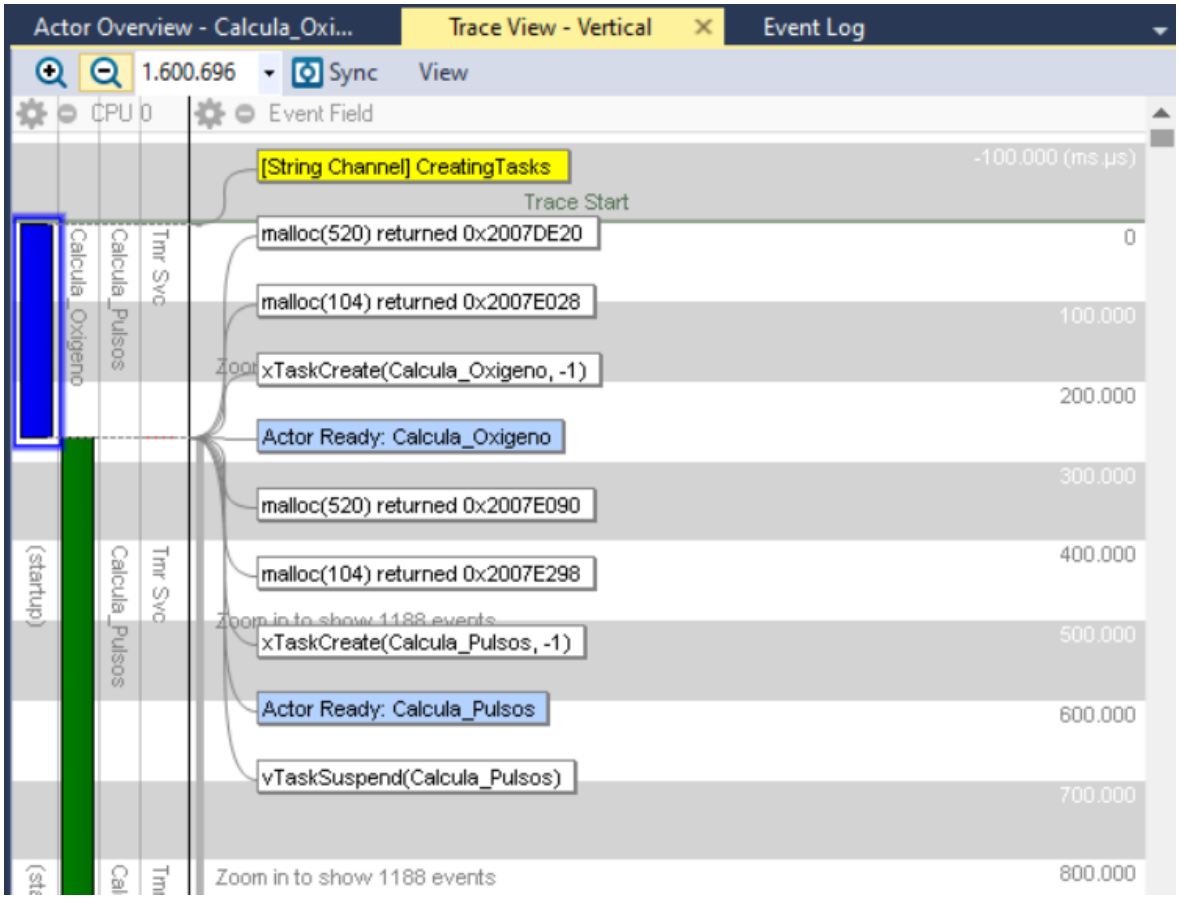



Figura 43. Recurso Trace View – Vertical indicando los eventos que ocurren dentro de *startup*

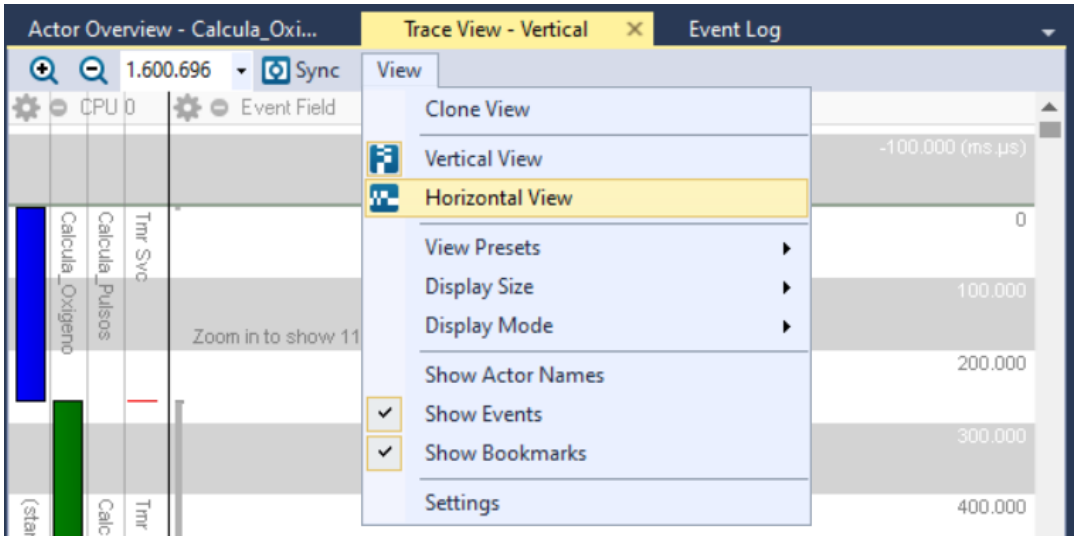



Figura 44. Selección de Vista Horizontal

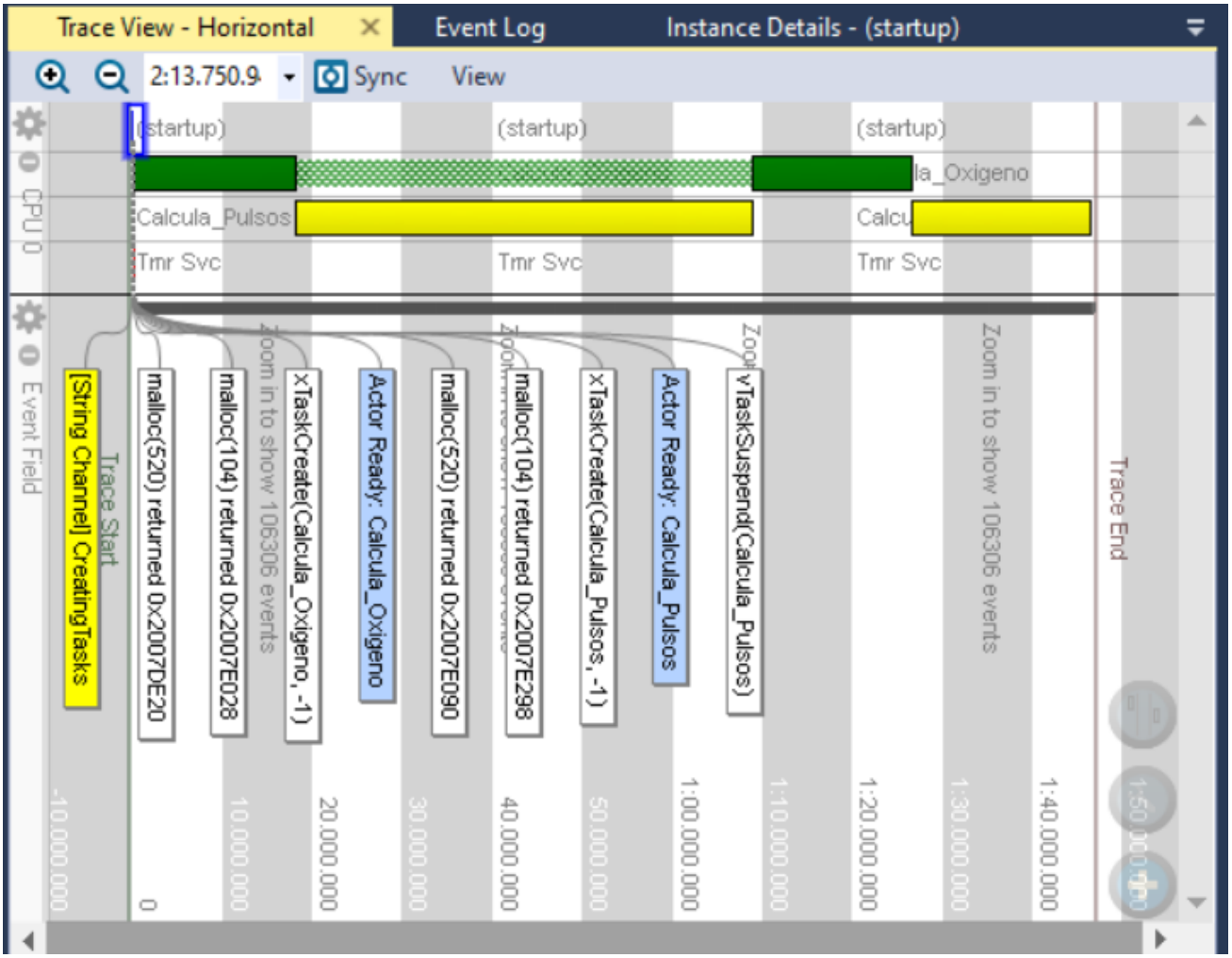



Figura 45. *Trace View – Horizontal* mostrando la primera instancia de la tarea 3

## 8.6. Instance Details

Cuando se hace doble clic en una instancia desde cualquier ventana, se redirecciona a la ventana ***Instance Details,*** donde se detalla lo siguiente. La figura 46 muestra el resultado de hacer doble clic en la primera instancia de la tarea ***Calcula_Pulsos*** desde la ventana ***Trace View***. Desde esta ventana se detalla lo siguiente:

- Se trata de la tarea ***Calcula_Pulsos***.
- Es la primera instancia de las 2 capturadas.
- Empieza en el tiempo 18.284.616 *(s.ms.us)* y finaliza en el tiempo 1:08.945.100 *(m : s.ms.us)*
- La siguiente instancia ocurre en el tiempo 1:26.601.616 *(ms.us)*
- La instancia tiene un tiempo de ejecución de 50.660.468 *(s.ms.us)*
- El número de fragmentos que tiene la instancia es 1. Es lógico porque la tarea ***Calcula_Pulsos*** es la más prioritaria y ninguna otra la puede interrumpir durante su ejecución.
- El uso total de la tarea ***Calcula_Pulsos*** en la CPU es de 47.5 %.
- La prioridad de la tarea ***Calcula_Pulsos*** es de valor igual a 3.
- Se han registrado 6 eventos:

- el primero es el mensaje *Calcula_Pulsos* para indicar que se ejecuta dicha tarea enviado a través de ***String Channel*** en el tiempo 18.284.638 *(s.ms.us)*
- El segundo evento es el envío del nivel de pulsos por minuto a través del canal ***PPM***
- El tercer evento es el envío del nivel de pulsos por minuto a través del canal ***PPM***
- El cuarto evento es el envío de la fecha y hora actual a través del canal ***Clock***
- El quinto evento es el envío del nivel de pulsos por minuto a través del canal ***PPM***
- El sexto evento es la suspensión de la tarea ***Calcula_Pulsos***

Los cinco primeros eventos están relacionados con los canales de eventos de usuario y por esa razón están resaltados en amarillo. En concreto se trata del canal ***PPM, Clock y String Channel.***

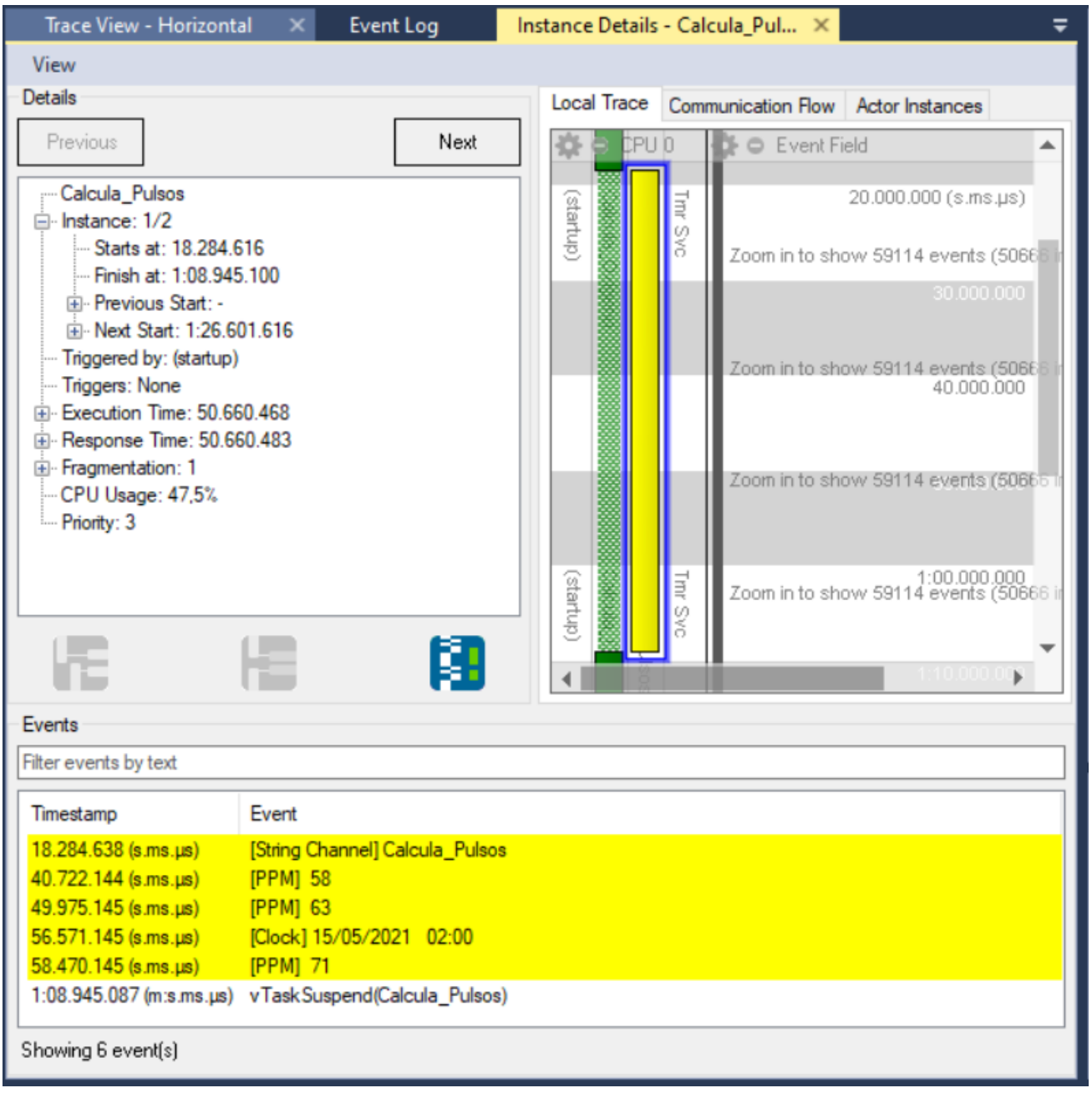



Figura 46. Instance Details mostrando la primera instancia de la tarea 3

Si se desea analizar la siguiente instancia no hay más que hacer clic sobre el botón ***Next***. Además, se puede extraer información adicional seleccionando la pestaña ***Actor Instances***. Aparece una lista con todas las instancias de la tarea 3 indicando:

- el inicio y fin de cada instancia,
- el número de fragmentos que tiene
- el tiempo de ejecución de cada una de ellas.

Si se selecciona una instancia en especial, en la parte inferior se indican los eventos ocurridos en dicha instancia.

Por ejemplo, en la figura 47, se ha seleccionado la primera instancia de la tarea ***Calcula_Pulsos*** y se puede extraer la siguiente información.

- Inicia en el tiempo 18.284.616 *(s.ms.us),* finaliza en el tiempo 1:08.945.100 *(m : s.ms.us)*
- tiene un tiempo de ejecución de 50.660.468 *(s.ms.us)*
- La instancia tiene un solo fragmento.

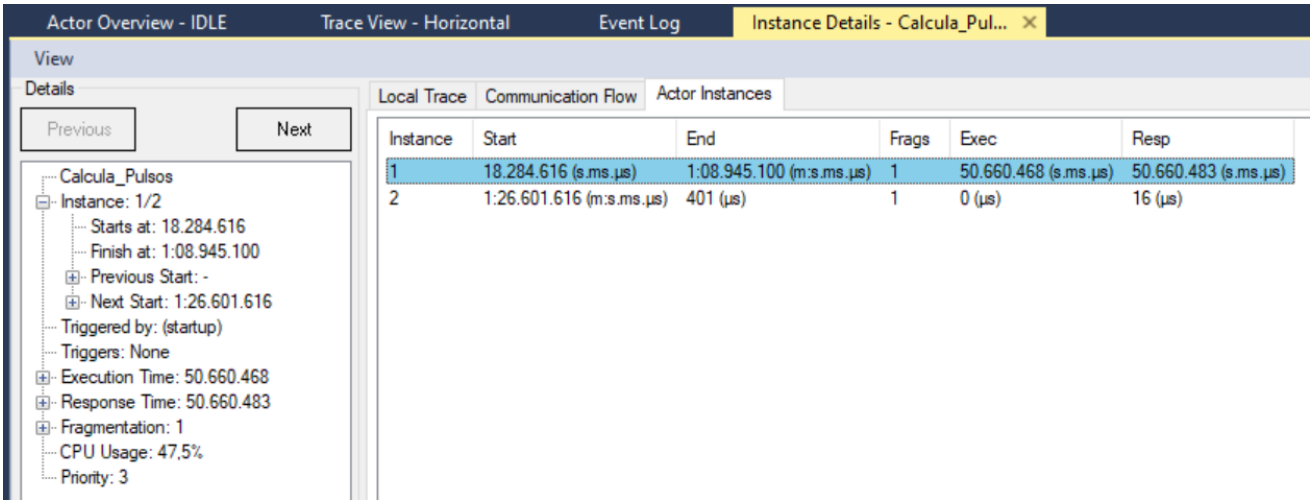



Figura 47. Pestaña *Actor Instances* de la ventana *Instance Details*

## 8.7. Finder – Quick Finder – Full Finder

La ventana ***Finder*** permite encontrar rápidamente instancias, llamadas a servicios y eventos de usuario empleando varios filtros. Además, permite saltar a un punto temporal de la traza capturada si se desea. Es un buscador realmente potente.

Se divide en dos tipos de buscadores:

- *Quick Finder*
- *Full Finder*

Ambos permiten realizar cualquier búsqueda en ***Tracealyzer*** como eventos, actores, vistas, incluso buscar en la ayuda del manual de usuario del propio ***Tracealyzer*** (también disponible pulsado F1). Los eventos son encontrados con su respectivo mensaje de texto incluyendo el nombre de los canales de eventos de usuario.

Cuando se combinan múltiples términos en una búsqueda, el resultado incluye cualquier palabra que se asemeje al término de la búsqueda, poniendo en primer lugar la opción que más se asemeje.

- ***Quick Finder*** es un buscador pequeño en forma de barra que se abre en cualquier ventana de ***Tracealyzer***. Es una manera rápida de hacer una búsqueda, pero solo muestra una cantidad limitada de resultados. La manera más rápida de abrir esta opción es pulsando ***Ctrl + F***. Existe un caso especial de esta opción y es **pulsando *Ctrl + G*** para acceder a ***Quick Finder*** pero solo para realizar saltos temporales. Estos atajos de teclado se pueden modificar desde la ventana ***Keyboard Mapping***. Por ejemplo, desde ***Trace View*** se ha pulsado ***Ctrl+F*** para abrir el buscador ***Quick Finder*** y se ha escrito "calcula_Pulsos" obteniendo los siguientes resultados en la siguiente figura.

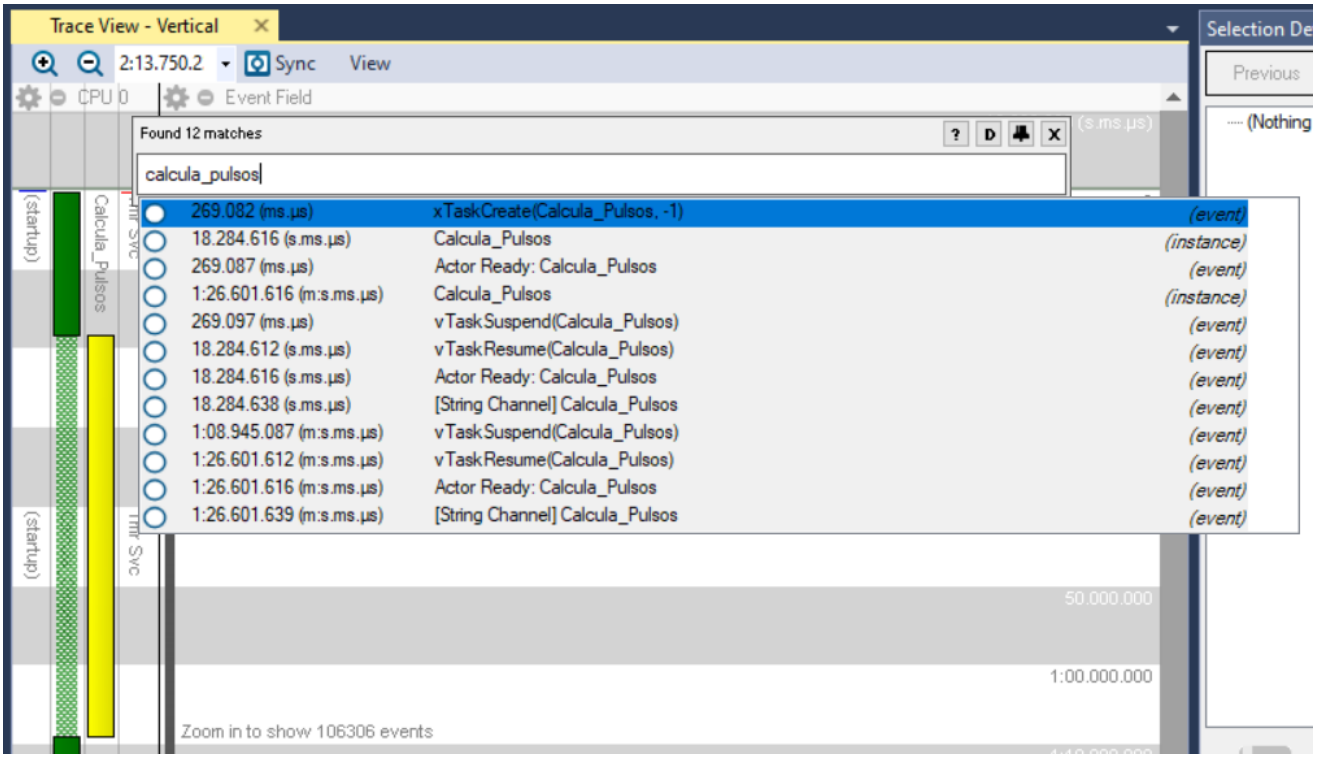



Figura 48. Ejemplo de Quick Finder buscando task3 con resultados limitados

A la izquierda de la lista se muestra el tiempo en el que ocurren los distintos resultados, en la parte central se muestra el nombre del resultado y a la derecha se especifica si se trata de una instancia o un evento. Dentro de la búsqueda realizada se ha seleccionado la primera entrada, la creación de la tarea ***Calcula_Pulsos***, se trata de un evento que ocurre en el tiempo 269.082 (ms.us) y haciendo doble clic nos lleva a la ventana ***Trace View*** donde se puede observar el evento resaltado en un recuadro azul., como en la figura 49.

Por otro lado, a modo de ejemplo, se ha abierto ***Quick Finder*** pulsando ***Ctrl+G*** para realizar un salto temporal. Se ha escrito el tiempo 119 800 000 us que corresponde con el final de la traza capturada como se muestra en la figura 49.

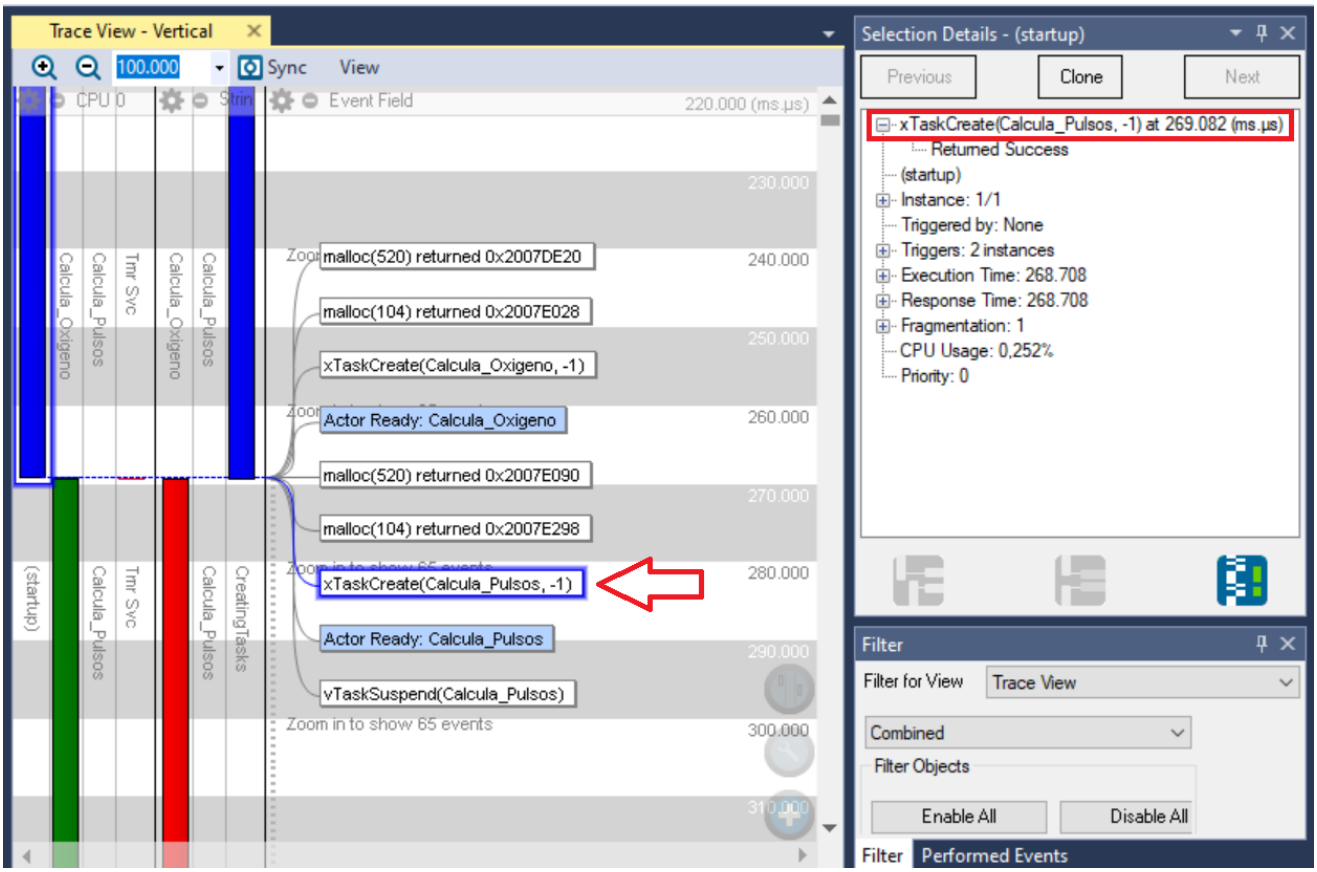



Figura 49. Trace View resaltando en azul la creación de la tarea Calcula_Pulsos desde *Quick Finder*

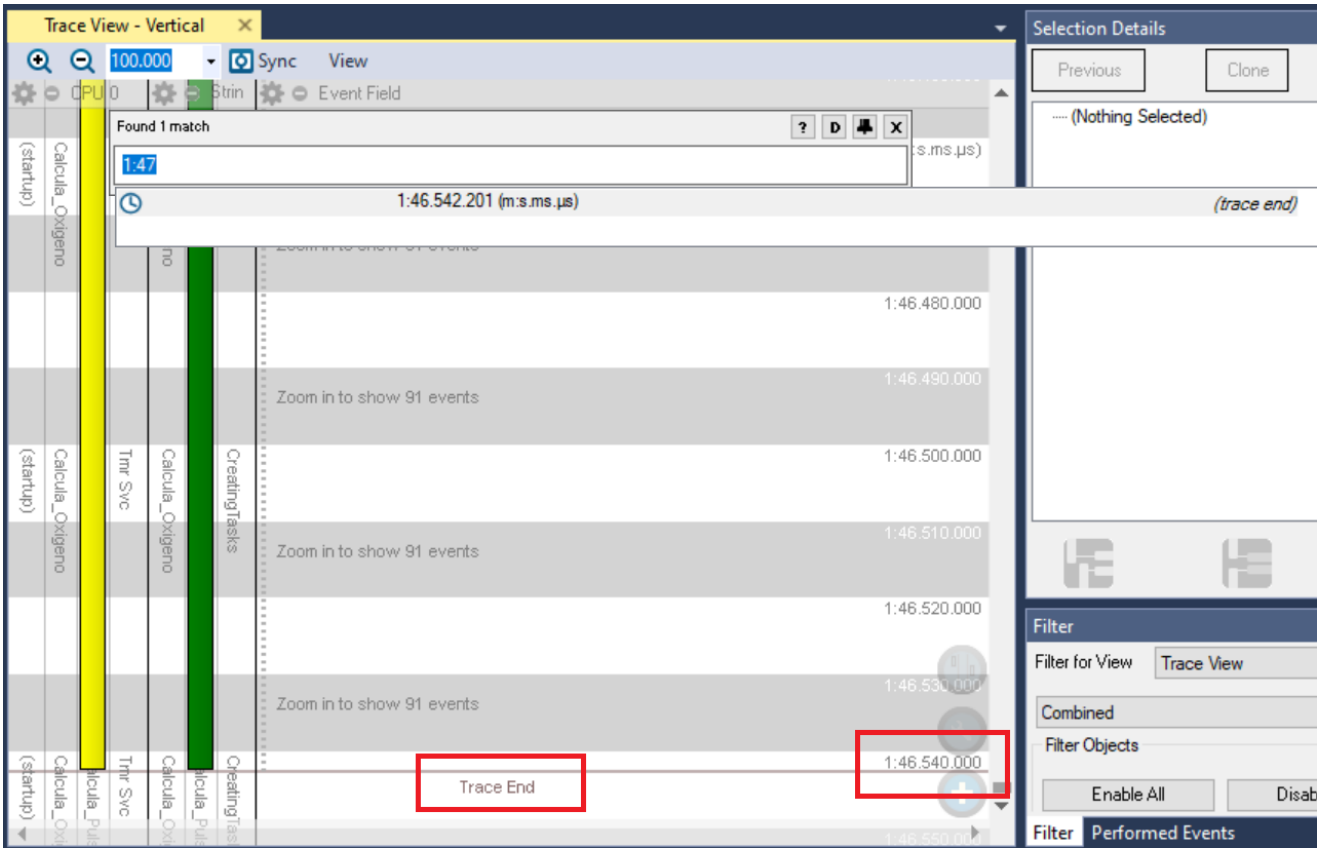



Figura 50. Salto temporal hacia el fin de la traza desde *Quick Finder* en la ventana *Trace View*

- ***Full Finder*** se puede abrir desde el menú de ventanas o pulsando ***Ctrl + Shift + F***. La diferencia con ***Quick Finder*** reside en que esta opción realiza una búsqueda completa mostrando todos los posibles resultados en una larga lista. En la figura 51 se ha realizado la búsqueda de "Calcula_Pulsos" y se obtiene una búsqueda completa, desde la creación de la tarea y su suspensión, hasta los mensajes enviados a través de ***String Channel***.

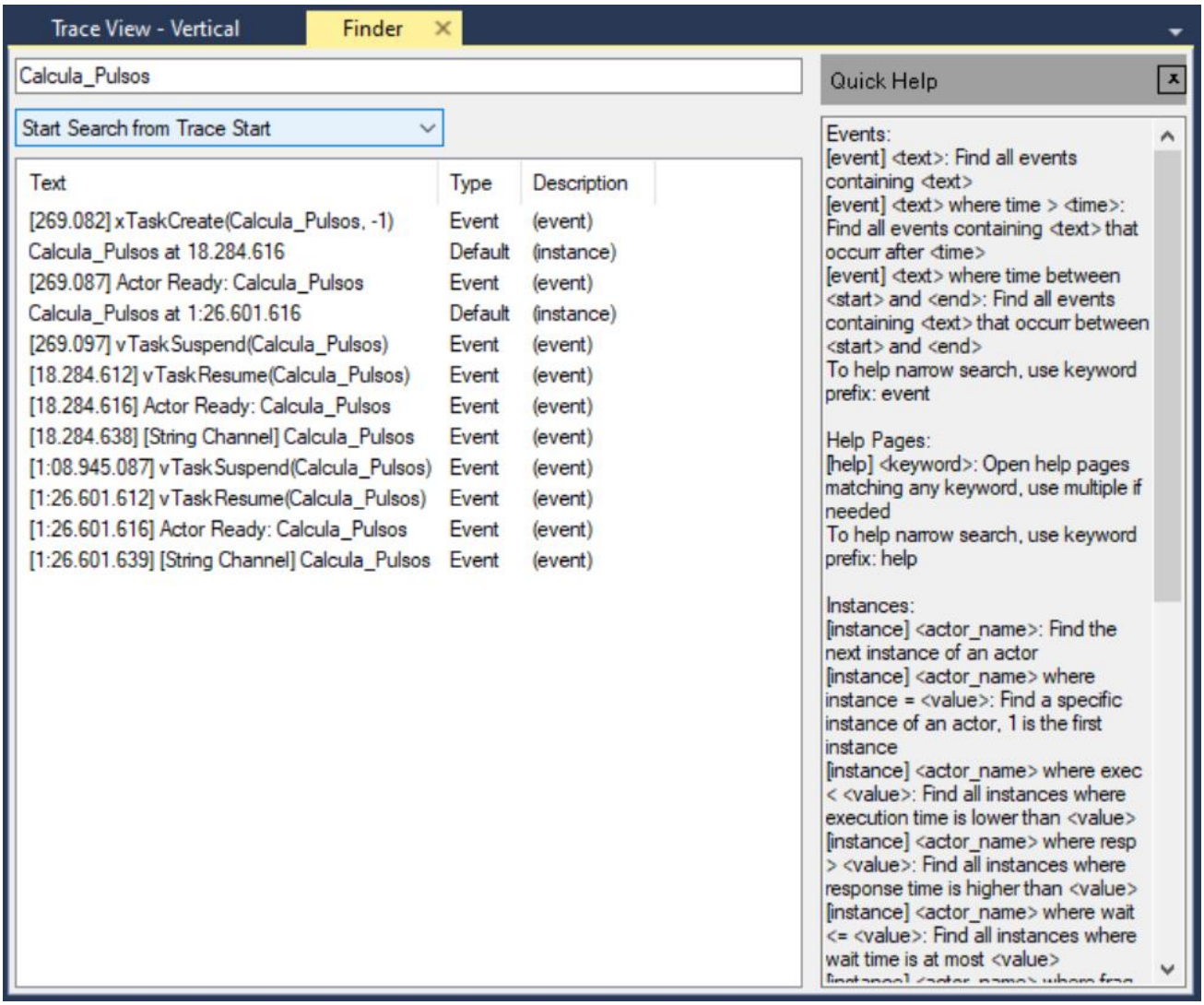



Figura 51. Ejemplo de Full Finder buscando task3 visualizando todos los resultados completos

*Finder* emplea un filtro de palabras clave que permite realizar búsquedas más específicas. Dichas palabras clave son:

- **Start** – inicio de una instancia
- **Ready** – tiempo que una instancia está preparada
- **End** – final de una instancia
- **Timestamp** – lo mismo que start
- **Instance** – el número de la instancia, comenzando por 1
- **Execution** – el tiempo de ejecución de una instancia
- **Wait** – el tiempo de espera de una instancia

- **Fragments** – el número de fragmentos de una instancia
- **Period** – el periodo de una instancia (tiempo desde el inicio una instancia previa al inicio de la instancia actual)
- **Separation** – la separación de una instancia (tiempo desde el fin de una instancia previa al inicio de la instancia actual)

No es necesario emplear las palabras clave por completo, se pueden acortar a la mitad de la palabra, por ejemplo, se puede escribir “exec” en lugar de “execution”.

Las palabras clave se pueden comparar con valores decimales empelando el símbolo “mayor que”, >, y “menor que”, <, además, se puede hacer una búsqueda de un tiempo máximo o mínimo escribiendo “max” o “min”.

Existe una cláusula llamada “where” que se emplea para realizar búsquedas más concretas, por ejemplo, una búsqueda de la tarea ***Calcula_Pulsos*** donde el tiempo de ejecución sea el más largo, la búsqueda tendría que ser ***Actor Calcula_Pulsos where exec is max***. Del apartado anterior ***Instance Details,*** de la pestaña ***Actor instances*** de la figura 47 se sabe que la instancia que tiene mayor duración es la instancia número 1. El resultado de la búsqueda en ***Quick Finder*** y ***Full Finder*** coincide con los datos esperados y se muestran en las siguientes figuras.

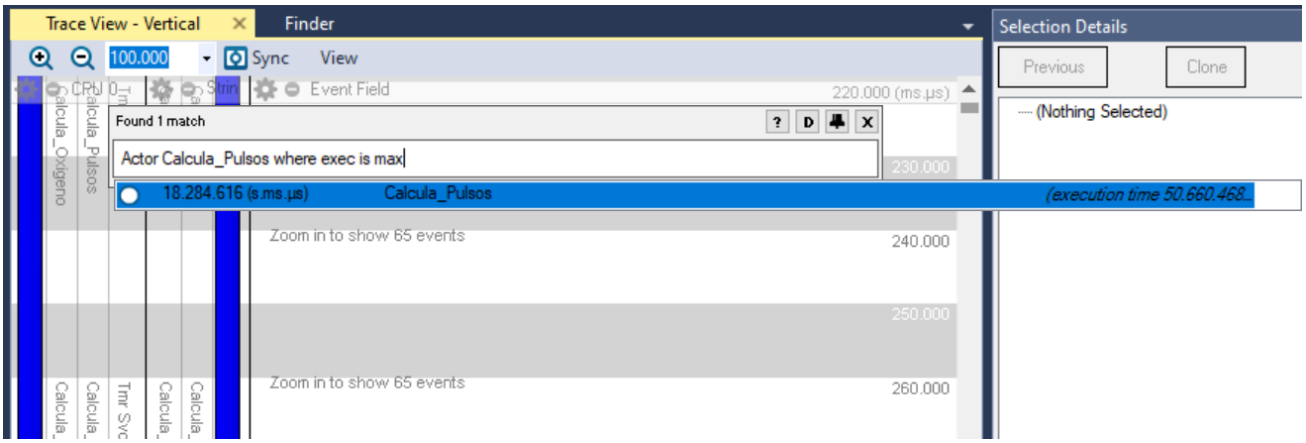



Figura 52. Resultado de la búsqueda “actor Calcula_Pulsos where exec is max” en *Quick Finder*

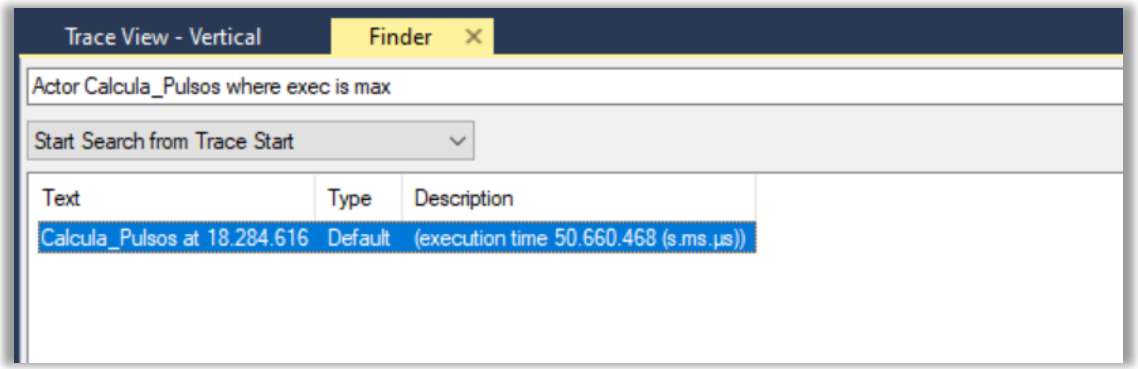



Figura 53. Resultado de la búsqueda “actor Calcula_Pulsos where exec is max” en *Full Finder*

Los tiempos se consideran del orden de microsegundos, por lo que si por ejemplo se desease saltar hacia el tiempo 200 (ms) desde la ventana ***Trace* View** se debe introducir ***time 200000*** en ***Quick Finder*** dando como resultado un salto temporal a dicho tiempo, como en la siguiente figura.

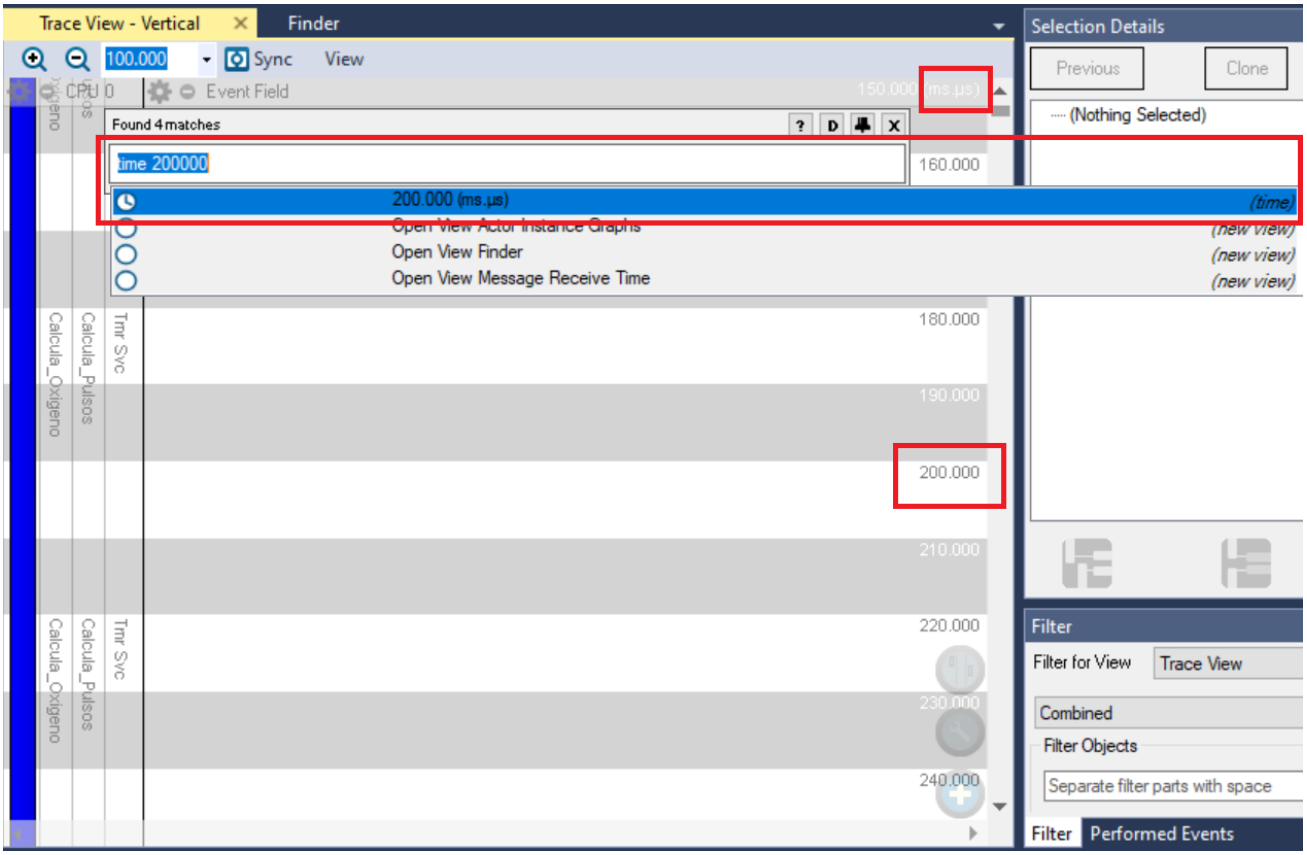



Figura 54. Salto temporal a 200 ms desde Trace View empleando Quick Finder

Otro ejemplo puede ser la búsqueda del evento ***PPM*** que envía el mensaje de pulsos por minuto a través de ***Tracealyzer*** filtrando además todos los que ocurran a partir de 1 segundo. El resultado son los eventos que ocurren en un tiempo superior a 1 segundo y se muestran en la siguiente figura.

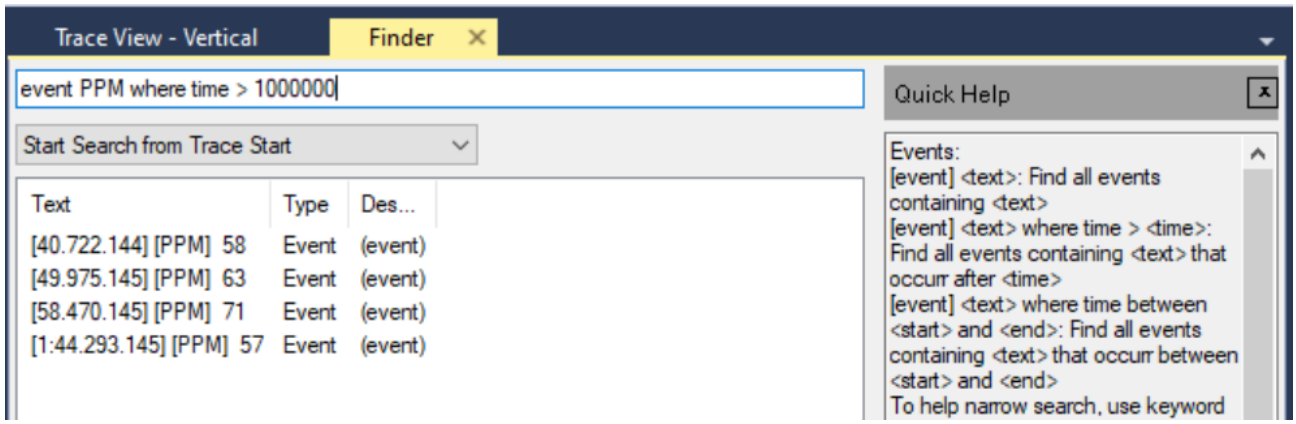



Figura 55. búsqueda del evento BPM a partir de 1 segundo con Full Finder

## 8.8. User Event Signal Plot

Esta ventana permite representar de manera gráfica los datos de los canales de eventos usuario. En uno de los apartados anteriores llamado ***Eventos de Usuario*** se explicó que en este proyecto se crearon cuatro canales para transmitir información y son los siguientes:

- String Channel
- Clock
- PPM
- SPO2

Los dos últimos transmiten el valor del nivel oxímetro y de pulsos por minuto, por lo que la ventana ***User Event Signal Plot*** puede ser muy útil para representar estos valores visualmente.

En la siguiente figura se observa el nivel de oxígeno en sangre, **SpO2** (%) y los pulsos por minuto, (**PPM**). Si se desea tener información sobre un punto en específico se debe hacer clic sobre él.

En el eje de ordenadas se representa el valor numérico de la variable **SPO2** que representa el nivel de oxígeno en sangre del usuario en %, y se representa el valor numérico de la variable **PPM** que representa los pulsos por minuto que se mide en el usuario.

En el eje de abscisas se representa el tiempo a lo largo de la traza capturada.

Se representa en color azul el valor del nivel de oxígeno en sangre. Se representa en color verde el valor de los pulsos por minuto. Ambos canales se inician con un valor igual a cero y se produce un salto de valores en el eje de abscisas cuando obtienen un nuevo valor, tal y como se indica con las flechas rojas en la siguiente figura.

En la siguiente tabla se indica que el canal **SPO2 y PPM** parten desde el inicio con un valor igual a cero. Posteriormente se produce el cálculo del nivel de oxígeno en sangre, **SPO2**, obteniéndose un valor de 96 %, posteriormente se procede a calcular los pulsos por minuto obteniéndose un valor de 58, 63 y 71. Estos datos están representados en una gráfica temporal gracias al recurso ***User Event signal Plot*** en la figura 56.

Tabla 8. Valor de los canales **SPO2** y **PPM**

| Tiempo (s .ms.us) | Canal | Valor |
|---|---|---|
| 0 | SPO2 | 0 |
| 0 | PPM | 0 |
| 18.040.012 | SPO2 | 96 |
| 40.722.144 | PPM | 58 |
| 49.975.145 | PPM | 63 |
| 58.470.145 | PPM | 71 |

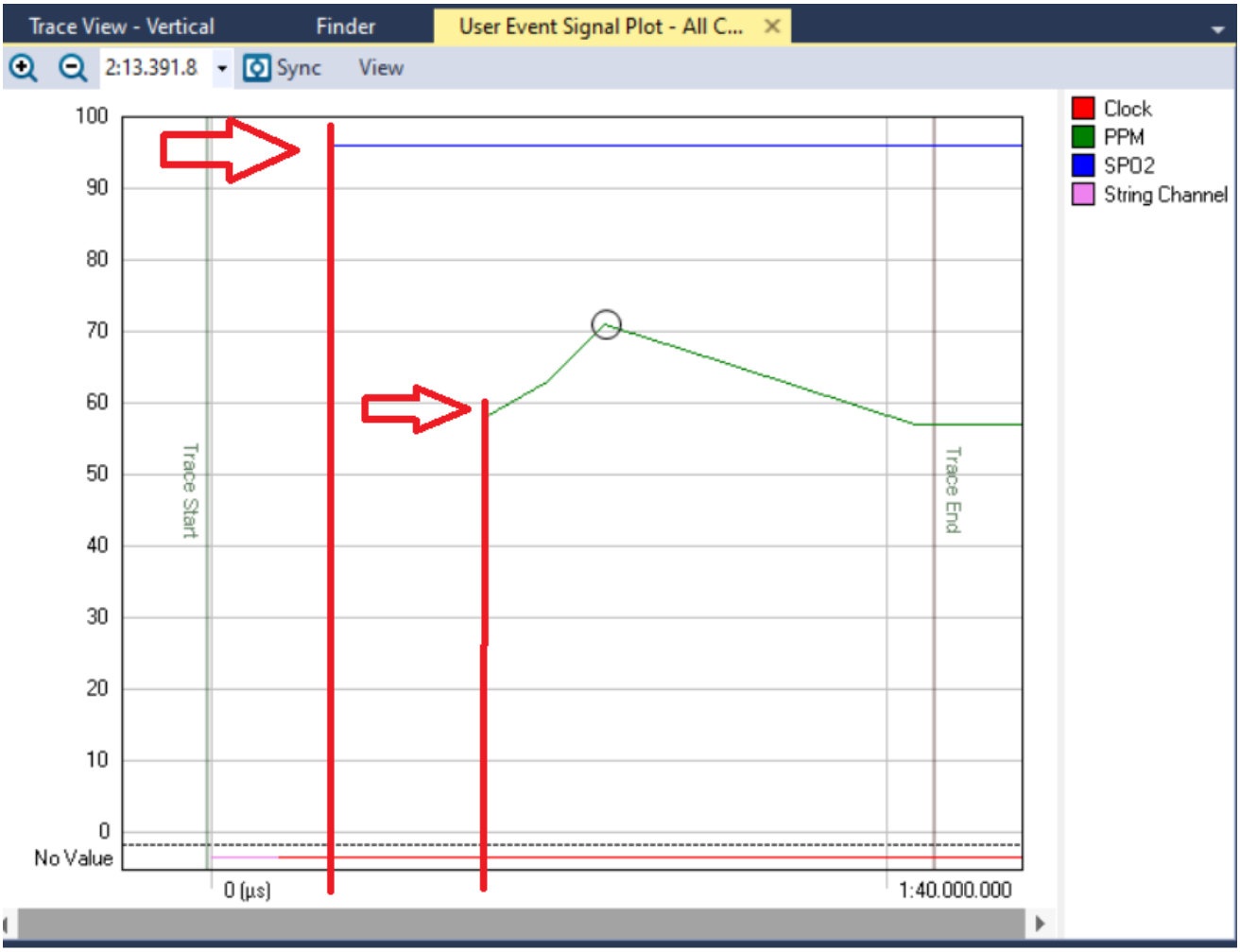



Figura 56. Gráfica del nivel de oxígeno en sangre y de los pulsos por minuto a través de Tracealyzer

## 8.9. Memory Heap Utilization

***Tracealyzer*** posee un recurso llamado ***Memory Heap Utilization*** que permite visualizar de manera gráfica la cantidad de memoria **Heap** que utiliza el sistema operativo.

En la siguiente figura se muestra de color rojo la memoria **Heap** del sistema. Es notable que se hace uso en 6 ocasiones de esta memoria, y corresponden con la reserva de memoria **Heap** al crear las tareas **Calcula_Oxígeno** y **Calcula_Pulsos**. En uno de los apartados anteriores llamado ***Event Log*** se ha explicado que el sistema operativo reserva en memoria 520 + 104 bytes cuando crea cada una de las tareas del usuario. Por esa razón existen cuatro puntos en la gráfica.

En total, el tamaño de memoria utilizado comprende entre 1200 y 1300 bytes, y siendo más precisos se puede saber que se utilizan 520 + 104 bytes por cada tarea, como son dos las tareas creadas, son 1248 bytes en total los que se utilizan.

Por ejemplo, se ha seleccionado el primer uso de memoria **Heap**, ocurre en el tiempo 268.983 (*ms.us*) durante la tarea ***startup***. La reserva de memoria se ha realizado correctamente (Returned Success). Esta información se puede ver en la parte derecha de la imagen, en la ventana pequeña llamada ***Selection Details***.


Figura 57. Utilización de la memoria Heap del sistema visualizada en forma gráfica

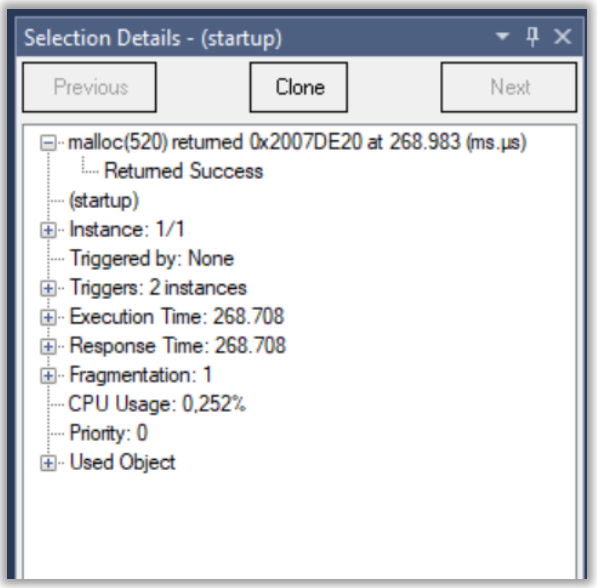



Figura 58. Selección del primer uso de la memoria HEAP

## 8.10. Intervals and State Machines

***Tracealyzer*** posee un potente recurso que permite modelar máquinas de estados a partir de los datos obtenidos en la traza capturada. Para conseguirlo es necesario seleccionar la opción ***Add Predefined***. Esta opción abre la ventana ***Predefined Intervals and States*** donde existen cinco grupos diferentes. El primer grupo, ***State Machines***, sirve para visualizar los estados que tienen las tareas (preparada, ejecutándose y en estado de espera). También permite ver la actividad de los núcleos del sistema. Los demás grupos no tienen repercusión acerca de las máquinas de estado por lo que se han dejado al margen del estudio.

Se han seleccionado a modo de ejemplo la tarea ***Calcula_Oxígeno***, ***Calcula_Pulsos*** y la actividad de los núcleos. Esto añade automáticamente el comportamiento de las tareas y de la **CPU** a la ventana ***Trace View*** y un diagrama de estados en la ventana ***State Machines Graph*** como en la figura 61, ventana que se tratará en el siguiente apartado. Como la CPU trabaja al 100% no permanece inactiva en ningún momento por lo que solo posee el estado de activo.

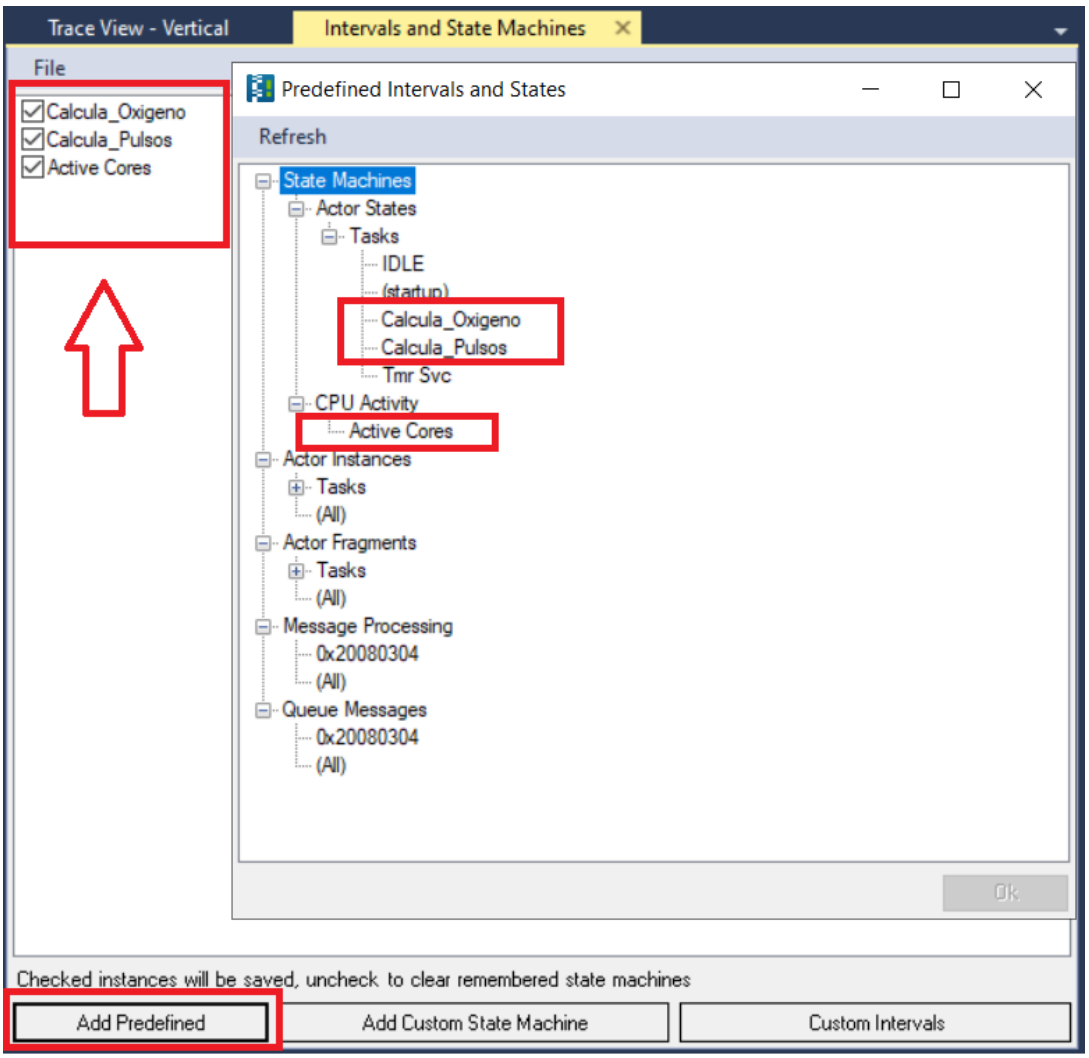



Figura 59. Selección de las tareas y de núcleos activos desde Predefined Intervals and States

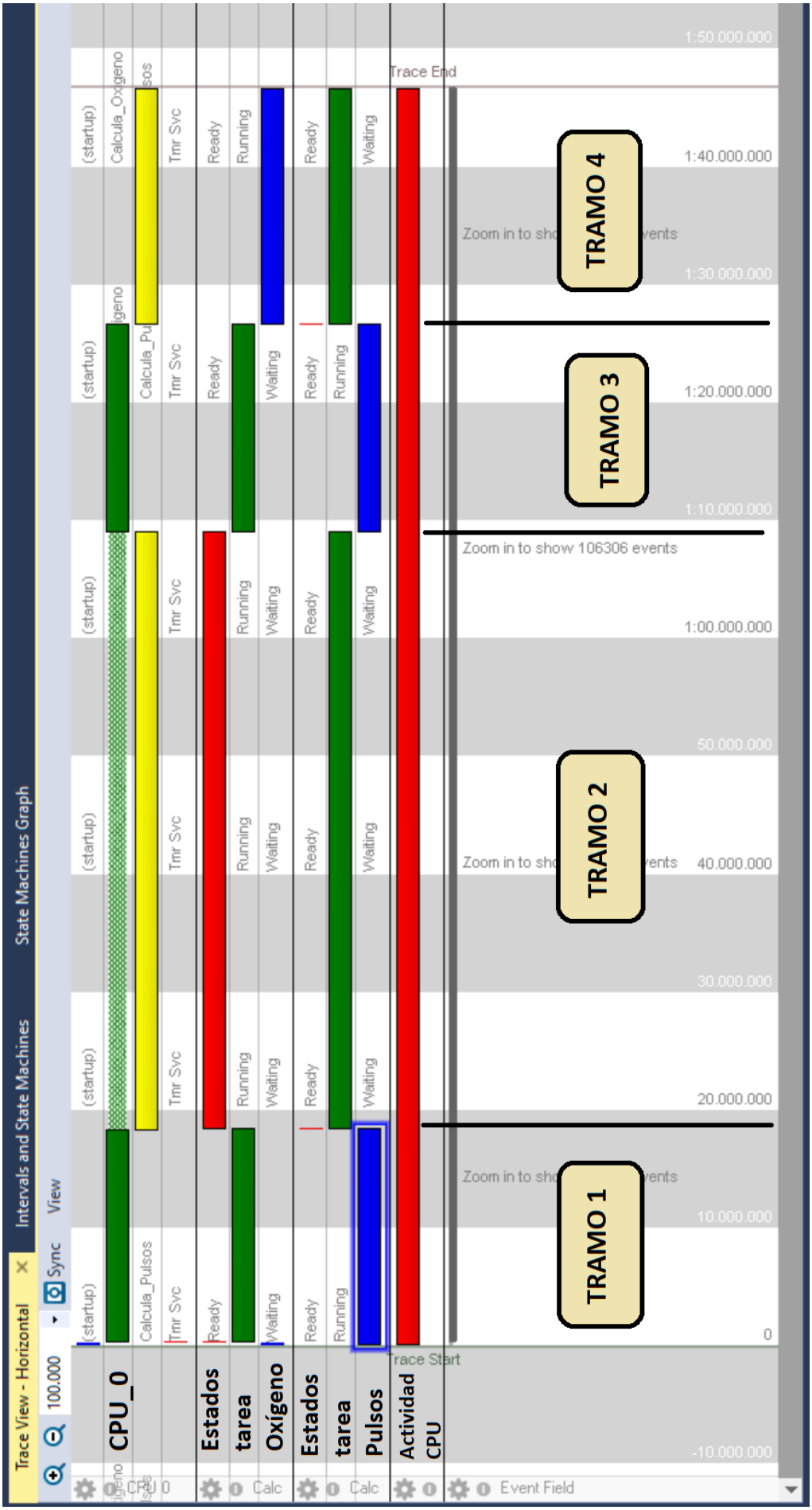



Figura 60. Ventana Trace View mostrando el comportamiento de las tareas y de la actividad de CPU

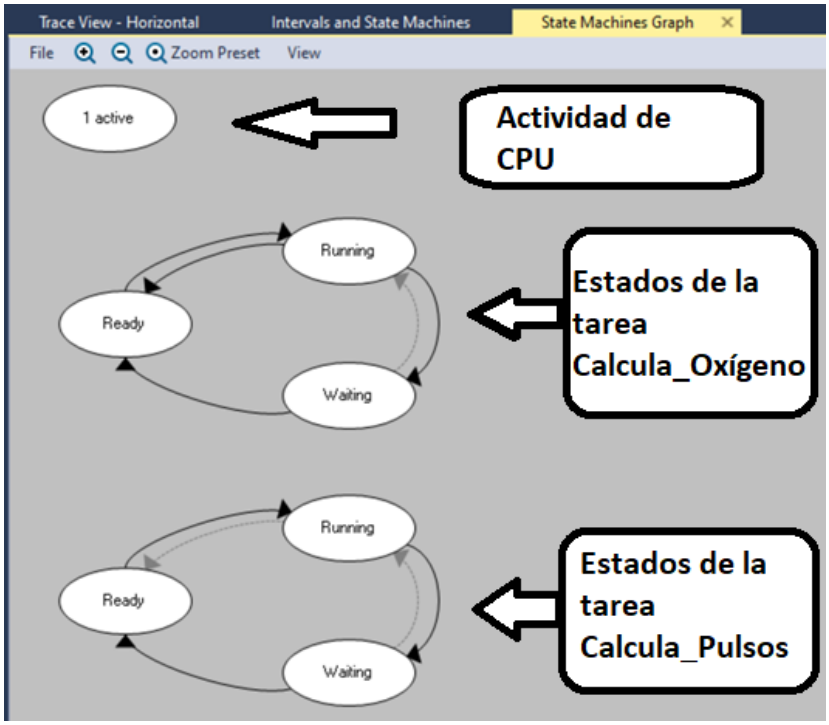



Figura 61. Máquina de estados para la actividad de CPU, tarea Calcula_Oxígeno y Calcula_Pulsos

A continuación, se analiza el comportamiento de las tareas del sistema y la actividad de CPU a partir de la figura 60. La figura 61 se analizará en el apartado ***State Machines Graph.***

- La zona marcada como **CPU_0** indica las tareas que se ejecutan en la CPU del sistema. En verde, la tarea ***Calcula_Oxígeno,*** en amarillo ***Calcula_Pulsos,*** en azul la tarea ***startup*** y en rojo la tarea ***Tmr Svc***.
- La zona marcada como ***Actividad de CPU*** indica cuando la CPU es utilizada y cuando no a lo largo del tiempo. Según la gráfica 61 la actividad de la CPU es constante, por lo que se representa en rojo cuando la CPU está en estado activo.
- Las zonas correspondientes a los estados de las tareas ***Calcula_Oxígeno y Calcula_Pulsos*** indican:
    - En color **rojo** cuando una tarea está preparada para ser ejecutada o ***READY***
    - En color **verde** cuando la tarea está ejecutándose o ***RUNNING***
    - En color **azul** cuando la tarea está en estado de espera o ***WAITING***

Según el comportamiento de las tareas se puede diferenciar 4 tramos en la traza capturada:

1. Por defecto, se ejecuta en primer lugar la tarea ***Calcula_Oxígeno***, obteniendo el *estado* ***RUNNING.*** La tarea ***Calcula_Pulsos*** y cualquier otra tarea en general, después de ser creada obtiene el estado de espera o ***WAITING***.
   Cuando la tarea ***Calcula_Oxígeno*** termina de realizar una medida de nivel de oxígeno, no se suspende, sino que lanza la función ***vTaskResume(Calcula_Pulsos)*** y la CPU pasa a ejecutar la tarea ***Calcula_Pulsos*** por tener mayor prioridad.

2. La tarea ***Calcula_Oxígeno*** es desalojada, por ese motivo se observa que deja de tener un color sólido y pasa a tener una textura como de una malla, y permanece en estado ***READY*** porque no se ha suspendido en ningún momento, está desalojada***.*** La tarea ***Calcula_Pulsos*** obtiene el estado de ***RUNNING***.

3. Cuando la tarea ***Calcula_Pulsos*** llega a su fin, se suspende a si misma con la función ***vTaskSuspend(Calcula_Pulsos)*** y obtiene el estado de ***WAITING***. La CPU retoma la ejecución de la tarea ***Calcula_Oxígeno***, y dicha tarea obtiene el estado de ***RUNNING***.

4. Por último, se vuelve a calcular un nuevo nivel de oxígeno y la tarea ***Calcula_Oxígeno*** ejecuta la función ***vTaskResume(Calcula_Pulsos)*** de manera que la tarea ***Calcula_Oxígeno*** debería tener el estado de ***READY***, pero como se ha comentado anteriormente, ***Tracealyzer*** en algunas ocasiones no consigue capturar correctamente la última instancia que se ejecuta antes de finalizar la traza, por ese motivo ***Tracealyzer*** interpreta que la tarea ***Calcula_Oxígeno*** está en estado ***WAITING*** cuando en realidad le corresponde el estado ***READY***.

   La tarea ***Calcula_Pulsos*** obtiene el estado de ***RUNNING*** hasta el fin de la traza.

   Durante todos los tramos la CPU permanece en estado activo representándose dicho estado de color rojo en la franja ***Actividad de CPU*** en la figura 60.

Por otro lado, se procede a analizar en detalle la ventana ***Intervals and State Machines,*** que ofrece cuatro diferentes opciones para cualquier tarea, haciendo clic en el botón izquierdo, obteniendo las opciones que se muestran en la siguiente figura, y son las siguientes:

- *Statistics* – Muestra un informe estadístico centrándose en la duración de cada estado.
- *Show timeline* – Muestra los estados sobre una línea temporal al igual que el recurso *Trace View*.
- *Create Inverted* – Crea un nuevo dato invertido. Por ejemplo, si la tarea seleccionada está en estado ejecutandose, el nuevo dato invertido mostrará un estado de espera y de preparado o ready.

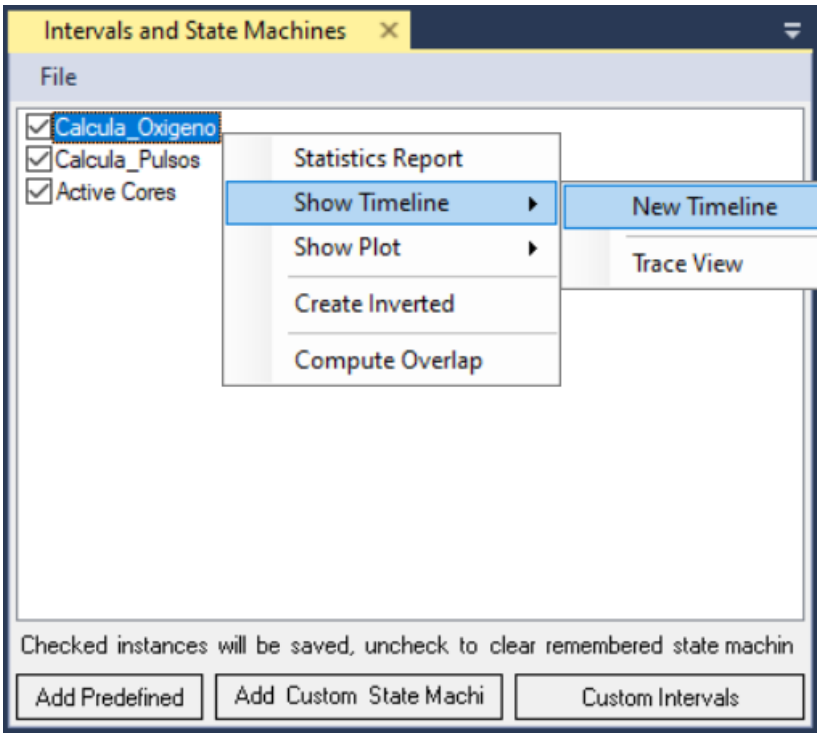



Figura 62. Opciones para cualquier selección de la ventana Invervals and State Machines

*Show Timeline*

Esta opción muestra los estados de cualquier tarea en forma de barras a lo largo del tiempo en la ventana ***Trace View*** desde la opción ***Show Timeline → Trace View*** como en la figura anterior. El resultado está remarcado en azul para la tarea ***Calcula_Oxígeno*** en la figura 63. Si se desea se puede abrir una ventana nueva en ***Trace View*** mostrando únicamente los estados de la tarea seleccionada desde la opción ***Show Timeline → New Timeline***. A modo de ejemplo se ha seleccionado una nueva ventana y el resultado queda reflejado en la figura 64.

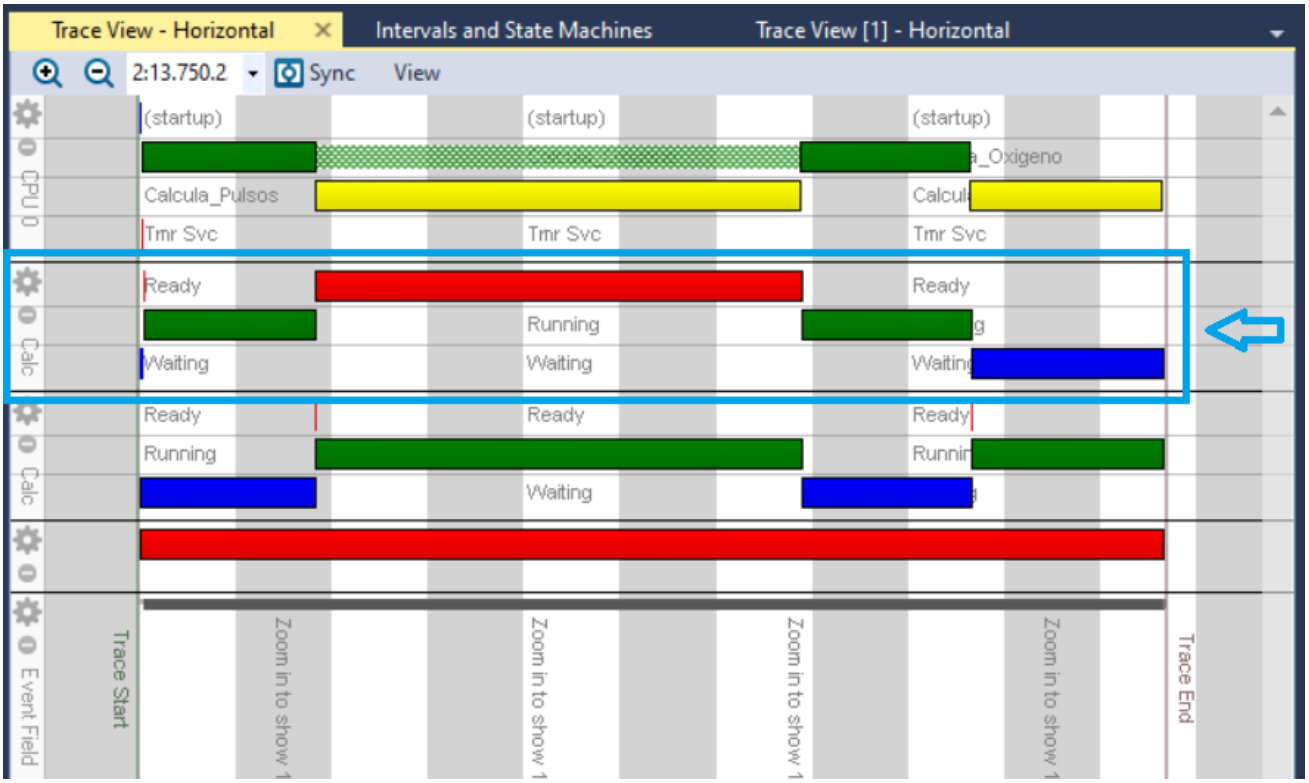



Figura 63. Representación gráfica de los estados de la tarea *Calcula_Pulsos* en Trace View

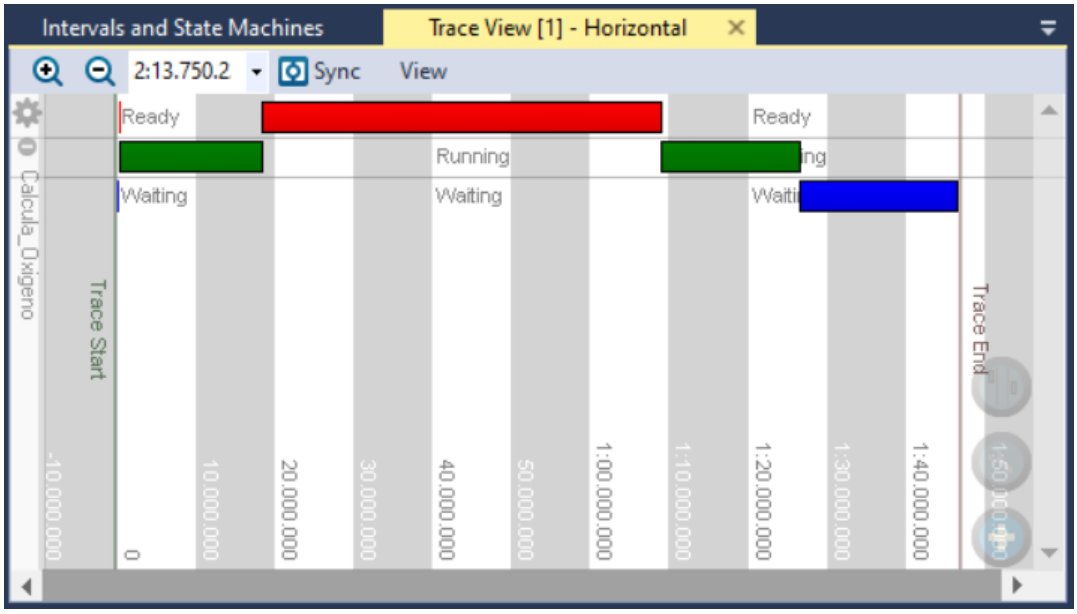



Figura 64. Estados de la tarea *Calcula_Oxígeno* en Trace View en una ventana nueva

*Create Inverted*

Esta opción crea una nueva entrada en la ventana ***Intervals and State MAchines*** como la tarea seleccionada, pero invertida. Por ejemplo, se ha creado la tarea ***Calcula_Oxígeno*** invertida obteniendo como resultado estados invertidos al real. La figura 65 muestra la nueva entrada creada, y la figura 66 muestra en la parte superior los estados de la tarea ***Calcula_Oxígeno*** y debajo, la inversión de dicha tarea. Se puede deducir que cuando en la parte superior la tarea está en estado activa o ***RUNNING***, debajo, la invertida, está en los estados opuestos, ***READY y RUNNING.*** Ocurre lo mismo con los demás estados.

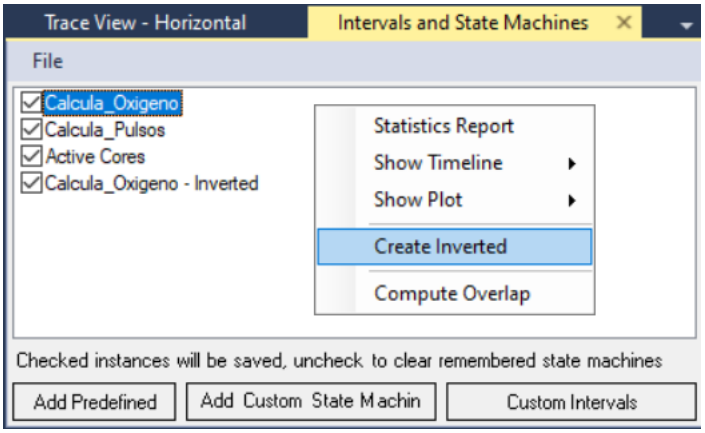



Figura 65. Nuevo dato en la ventana Intervals and State Machines

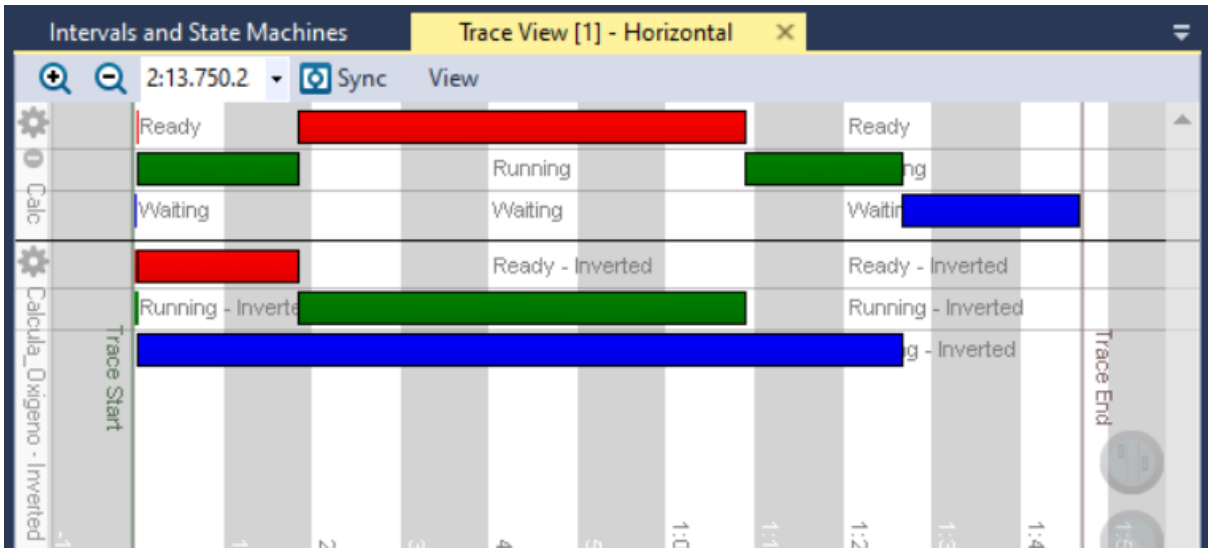



Figura 66. Trace View mostrando la tarea 1 arriba y la tarea 1 invertida debajo.

*Statistics*

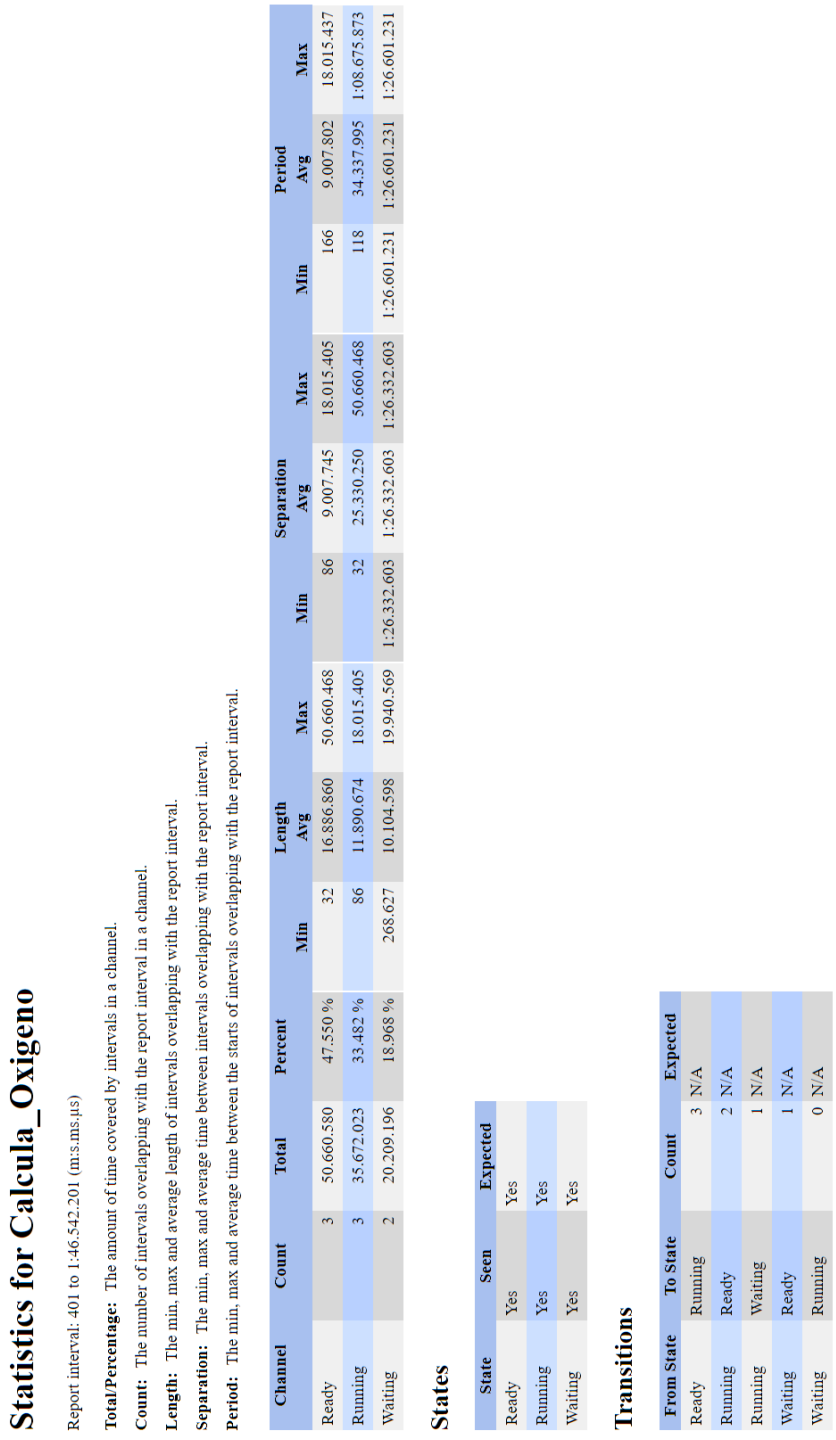
Para realizar una demostración de este recurso, se ha generado un informe estadístico de los estados de la tarea ***Calcula_Oxígeno*** durante la sesión de trazabilidad capturada. En la estadística generada se entiende a la tarea ***Calcula_Oxígeno*** como una unidad del 100% con un tiempo de ejecución de 35.672.023 (*s.ms.us*).

La tarea ***Calcula_Oxígeno*** ha permanecido en estado ***READY*** 3 veces durante 50.660.580 (s.*ms.us*) o lo que es lo mismo, un 47.55%, ha permanecido en estado ***RUNNING*** *3* veces también durante un 33.482% y en estado ***WAITING 2*** veces durante un 18.968%

El informe trabaja sobre tres tipos de tiempos:

- La longitud, es decir, el tiempo que dura el estado
- La separación entre dos estados iguales
- El periodo o en otras palabras el tiempo entre dos estados iguales comenzando a contar desde el inicio de los estados.

Para estos tres tipos de tiempos se ofrece el tiempo mínimo, el tiempo máximo y la media. En la otra tabla en la zona inferior se muestra las posibles transiciones entre estados y se cuentan las veces que ocurren ese tipo de transiciones.



**Statistics for Calcula_Oxigeno**

Report interval: 401 to 1:46.542.201 (m:s.ms.µs)

**Total/Percentage:** The amount of time covered by intervals in a channel.

**Count:** The number of intervals overlapping with the report interval in a channel.

**Length:** The min, max and average length of intervals overlapping with the report interval.

**Separation:** The min, max and average time between intervals overlapping with the report interval.

**Period:** The min, max and average time between the starts of intervals overlapping with the report interval.

| Channel | Count | Total | Percent | Length Min | Length Avg | Length Max | Separation Min | Separation Avg | Separation Max | Period Min | Period Avg | Period Max |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Ready | 3 | 50.660.580 | 47.550 % | 32 | 16.886.860 | 50.660.468 | 86 | 9.007.745 | 18.015.405 | 166 | 9.007.802 | 18.015.437 |
| Running | 3 | 35.672.023 | 33.482 % | 86 | 11.890.674 | 18.015.405 | 32 | 25.330.250 | 50.660.468 | 118 | 34.337.995 | 1:08.675.873 |
| Waiting | 2 | 20.209.196 | 18.968 % | 268.627 | 10.104.598 | 19.940.569 | 1:26.332.603 | 1:26.332.603 | 1:26.332.603 | 1:26.601.231 | 1:26.601.231 | 1:26.601.231 |

**States**

| State | Seen | Expected |
|---|---|---|
| Ready | Yes | |
| Running | Yes | |
| Waiting | Yes | |

**Transitions**

| From State | To State | Count | Expected |
|---|---|---|---|
| Ready | Running | 3 | N/A |
| Running | Ready | 2 | N/A |
| Running | Waiting | 1 | N/A |
| Waiting | Ready | 1 | N/A |
| Waiting | Running | 0 | N/A |

Figura 67. Informe estadístico para la tarea Calcula_Oxígeno en relación con sus estados

## 8.11. State Machines Graph

En esta ventana se pueden analizar las transiciones entre estados para verificar si se realizan de manera correcta o si existe algún tipo de bloqueo. Para visualizar cualquier máquina de estados es necesario añadir un predefinido desde la ventana ***Intervals and State Machines*** como se ha hecho previamente para la tareas ***Calcula_Oxígeno, Calcula_Pulsos*** y para la actividad de CPU. Pero analizar los estados de una tarea de manera individual no resulta de mucho interés, lo que realmente importa es conocer cómo se comportan las tres tareas del sistema de manera global y ver qué transiciones se dan entre ellas. Por eso se ha decidido analizar las tareas del sistema como estados y ver la interacción existente entre ellas.

Si se entiende cada tarea como un estado, se pueden analizar las transiciones entre tareas y ver de manera global cómo funciona el sistema. Para que ***Tracealyzer*** entienda a las tareas del sistema como estados se ha creado un canal llamado ***String Channel*** donde se escribe un mensaje indicando la tarea que va a ser ejecutada en la CPU. De esta forma se crean dos estados que representan las dos tareas del sistema.

Para poder añadirla es necesario seleccionar la opción ***Add Custom State Machines***, y dentro de esa opción seleccionar ***Simple,*** lo que nos permite añadir el canal ***String Channel*** a la ventana ***Intervals and State Machines***. En la figura 67 se detallan los pasos necesarios y la aparición de ***String Channel***.

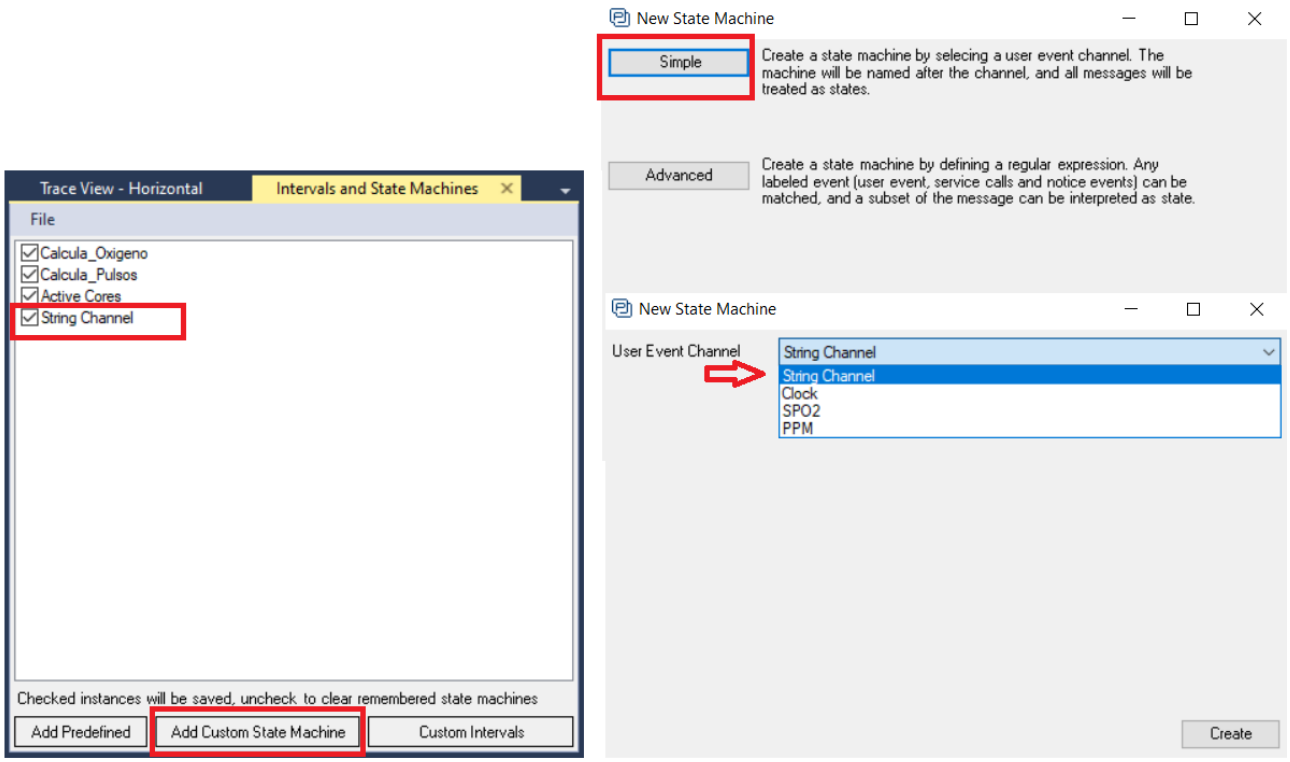



Figura 68. Resultado de añadir el canal String Channel a la ventana Intervals and State Machines

El resultado de crear la máquina de estados de las tareas del sistema y de la actividad de la CPU se refleja en la ventana ***State Machine Graph*** en la figura 71.

El estado ***CreatingTasks*** es un mensaje que se lanza cuando el sistema está inicializándose por lo que representa la tarea ***startup***, el estado ***Calcula_Oxígeno*** representa la tarea ***Calcula_Oxígeno***, el estado ***Calcula_Pulsos*** representa la tarea ***Calcula_Pulsos***. A primera vista se aprecia que las tareas interactúan entre

ellas, lo que indica un funcionamiento correcto. Se recuerda que la tarea ***Calcula_Pulsos*** tiene mayor prioridad respecto a la tarea ***Calcula_Oxígeno***, por eso se observan las flechas en ambas direcciones.

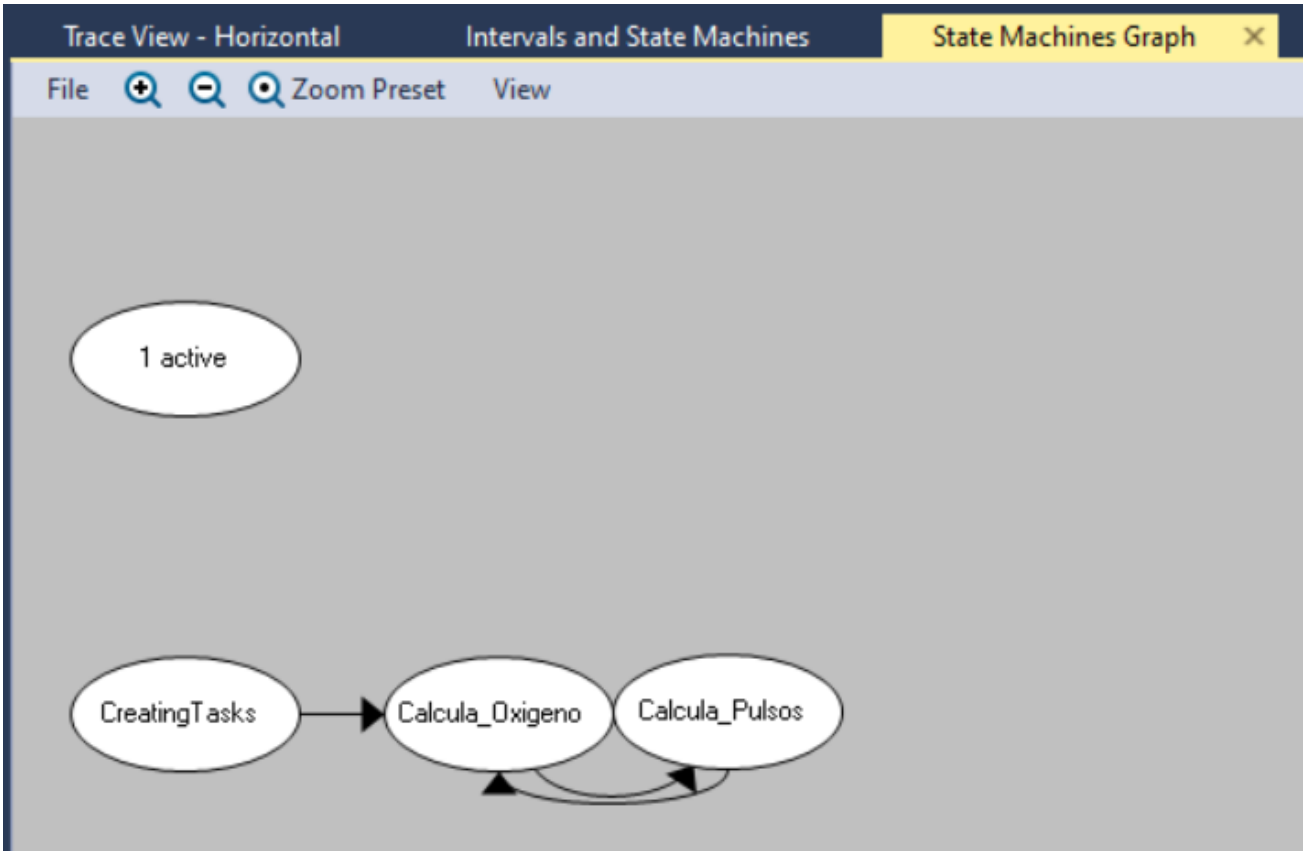



Figura 69. State Machine Graph con la máquina de estados de las tareas Calcula_Oxígeno y Calcula_Pulsos

Si se hace doble clic sobre un estado, se abre la ventana ***Interval Details***. A modo de ejemplo se ha seleccionado el estado ***Calcula_Oxígeno***. Esta acción nos abre la ventana ***Interval Details*** con los siguientes datos:

- En la parte de la izquierda se indica la duración del intervalo seleccionado, se indica que se trata del estado ***Calcula_Oxígeno*** aunque ***Tracelayzer*** lo entiende como Channel, cuenta las veces que la tarea permanece en este estado, que son dos, y además se muestra datos estadísticos acerca del estado seleccionado como en el apartado *Statistics*.
- En la pestaña ***Local Trace*** se muestra de manera gráfica el intervalo seleccionado junto a los eventos que suceden. Los eventos corresponden al canal ***String Channel*** que indica que se inicia la ejecución de la tarea ***Calcula_Oxígeno*** resaltado en amarillo.
- Si se desea seleccionar la siguiente vez que se ejecuta el estado ***Calcula_Oxígeno*** se debe clicar sobre ***NEXT***.
- Si se desea seleccionar el siguiente estado, que corresponde con la ejecución de la tarea ***Calcula_Pulsos*** se debe seleccionar ***NEXT STATE***.

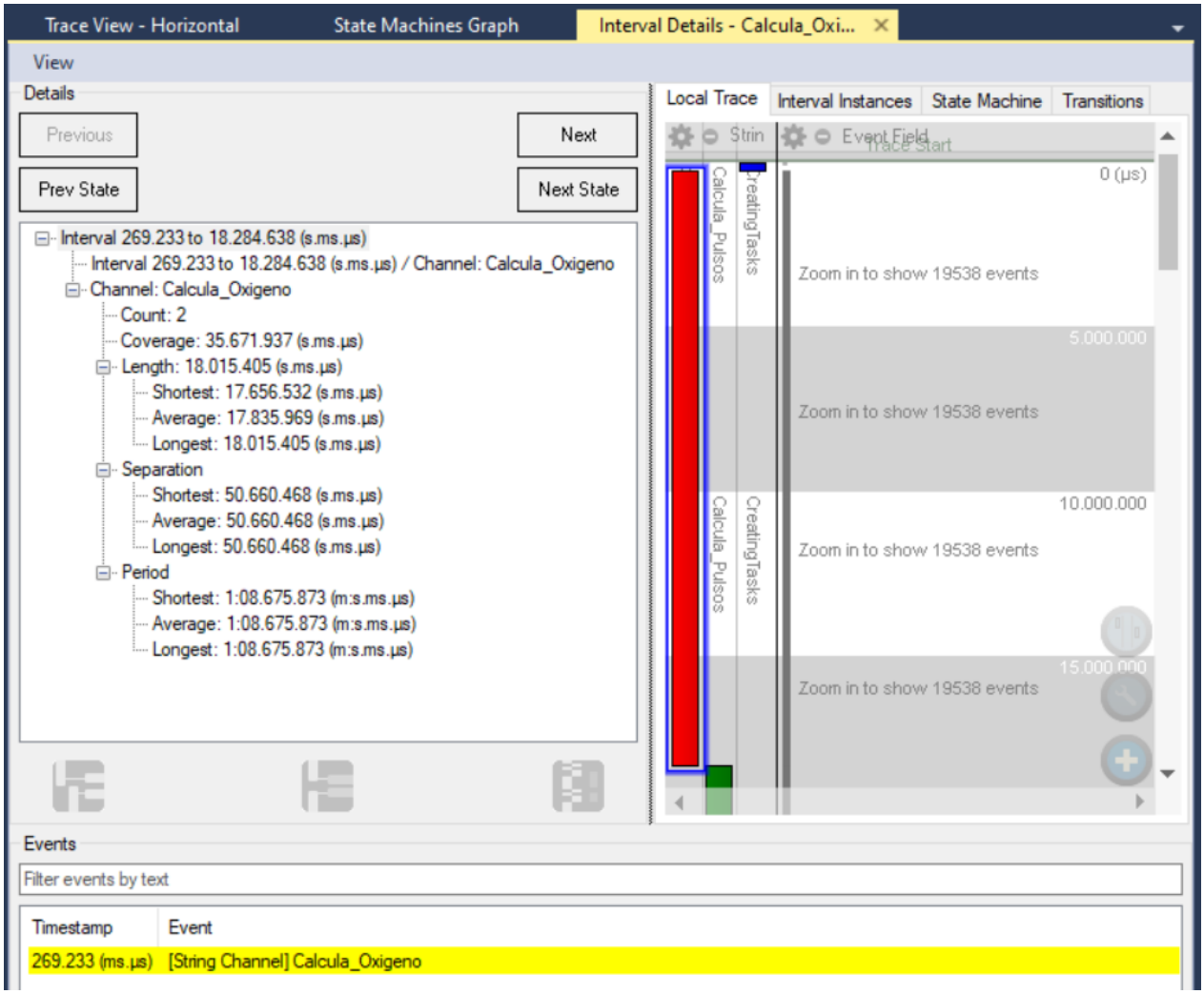



Figura 70. Invertal Details mostrando la opción local trace de Calcula_Oxígeno

Si se selecciona la pestaña ***Interval Instances*** como en la siguiente figura, se pueden ver todos los intervalos de todas las tareas. Se observa que la tarea ***Calcula_Oxígeno y Calcula_Pulsos*** se ejecutan dos veces siendo la primera enumerada con un cero y la segunda con un uno. Se especifica el inicio, fin y duración del intervalo. Si se hace doble clic en un intervalo se resalta dicho intervalo en la ventana ***Trace View***. Se ha seleccionado el intervalo ***Calcula_Oxígeno_0*** para visualizarlo en la ventana ***Trace View*** junto con la viste general de las tareas. Dicho intervalo seleccionado corresponde cuando la tarea ***startup*** finaliza y la tarea ***Calcula_Oxígeno*** está lista para ser ejecutada.

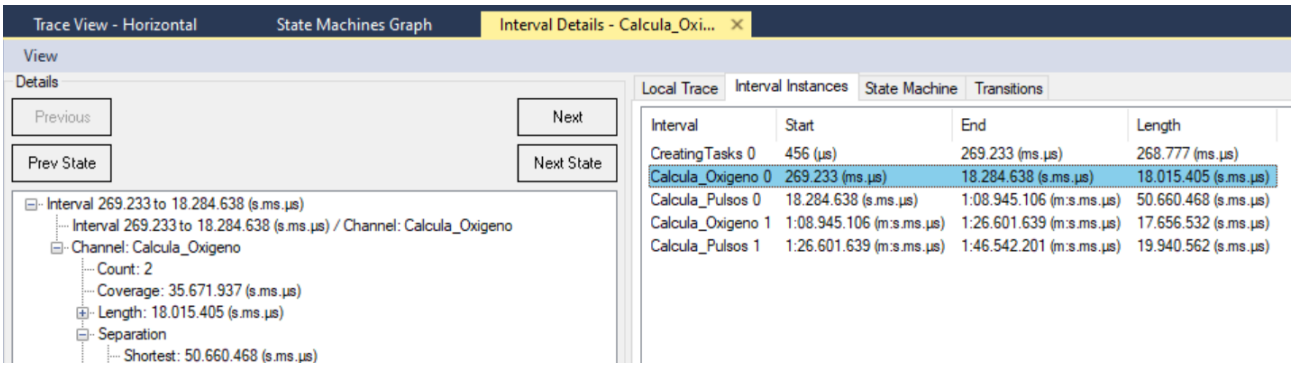



Figura 71. Pestaña *Interval Instances* desde la ventana *Instance Details*

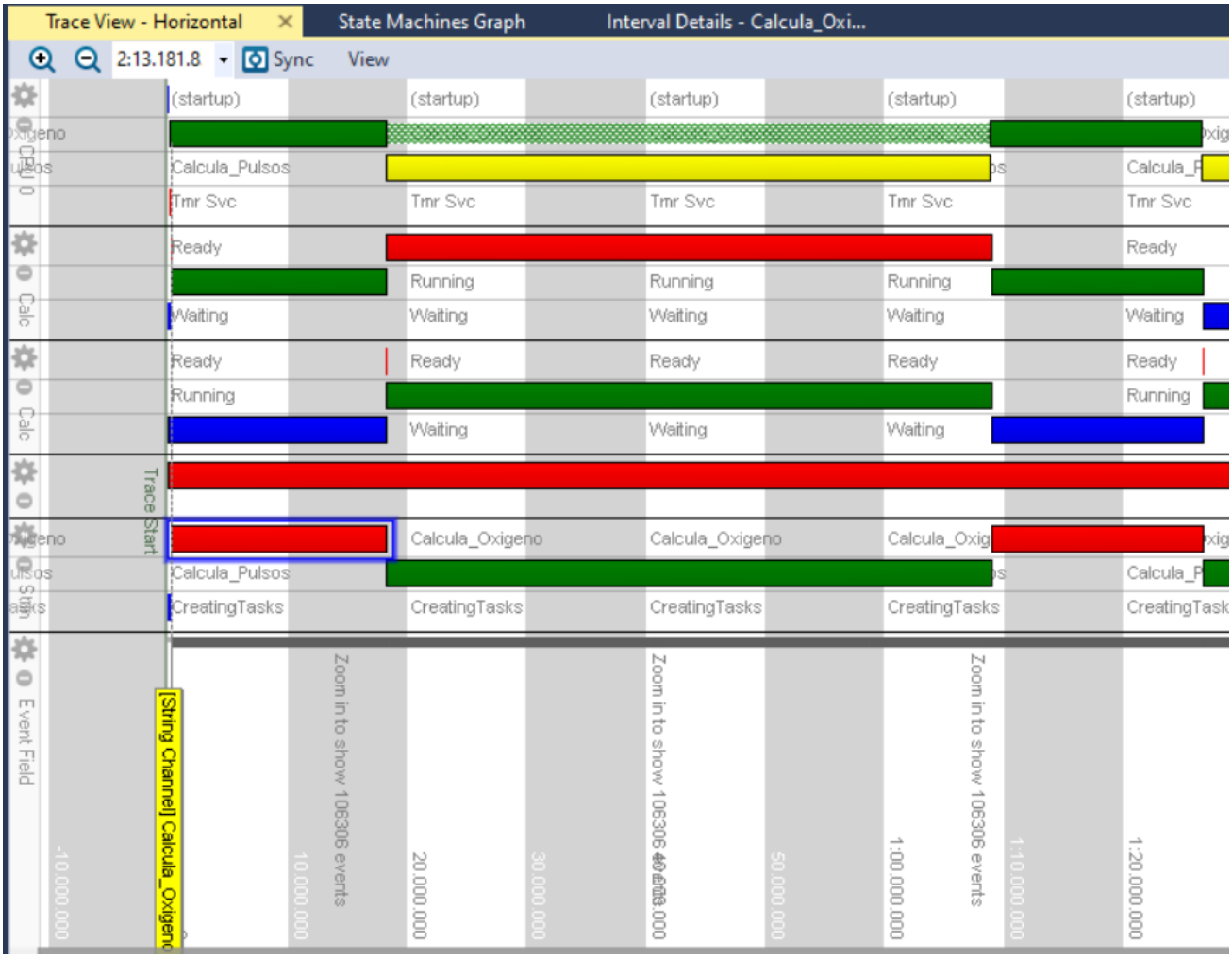



Figura 72. Selección del intervalo Calcula_Oxígeno 0 en la ventana Trace View

Por último, en la pestaña ***Transitions*** se pueden ver todas las transiciones entre todos los estados posibles y el momento en que se llevan a cabo. Posee un filtro donde se pueden seleccionar las transiciones desde un estado hacia otro en concreto. Además, si desea ver esa transición en la ventana ***Trace View*** se debe hacer doble clic. En la figura 75 se ha seleccionado la transición de ***Creating Tasks*** hacia ***Calcula_Oxígeno*** de manera que se pueda visualizar en la ventana ***Trace View.***

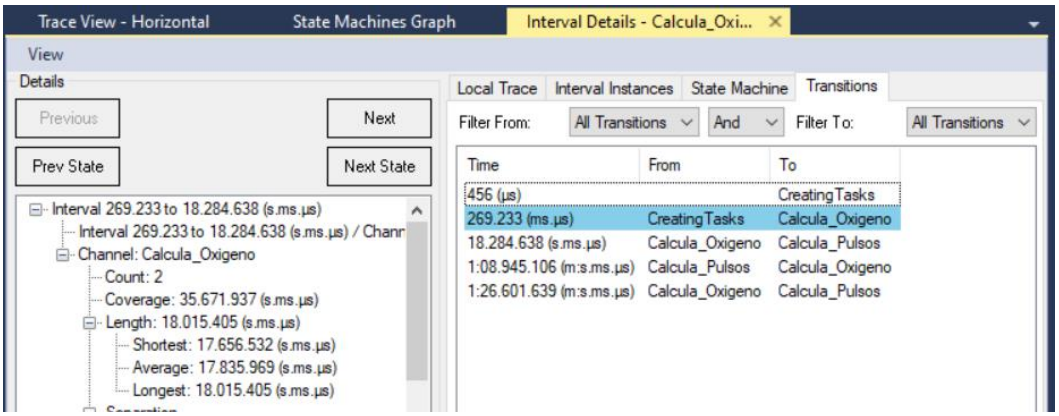



Figura 73. Pestaña T*ransitions* desde la ventana *Intervals Details*

El resultado de seleccionar la transición desde ***Creating Tasks*** hacia ***Calcula_Oxígeno*** se resalta en una línea azul como se señala con la flecha roja en la siguiente figura.

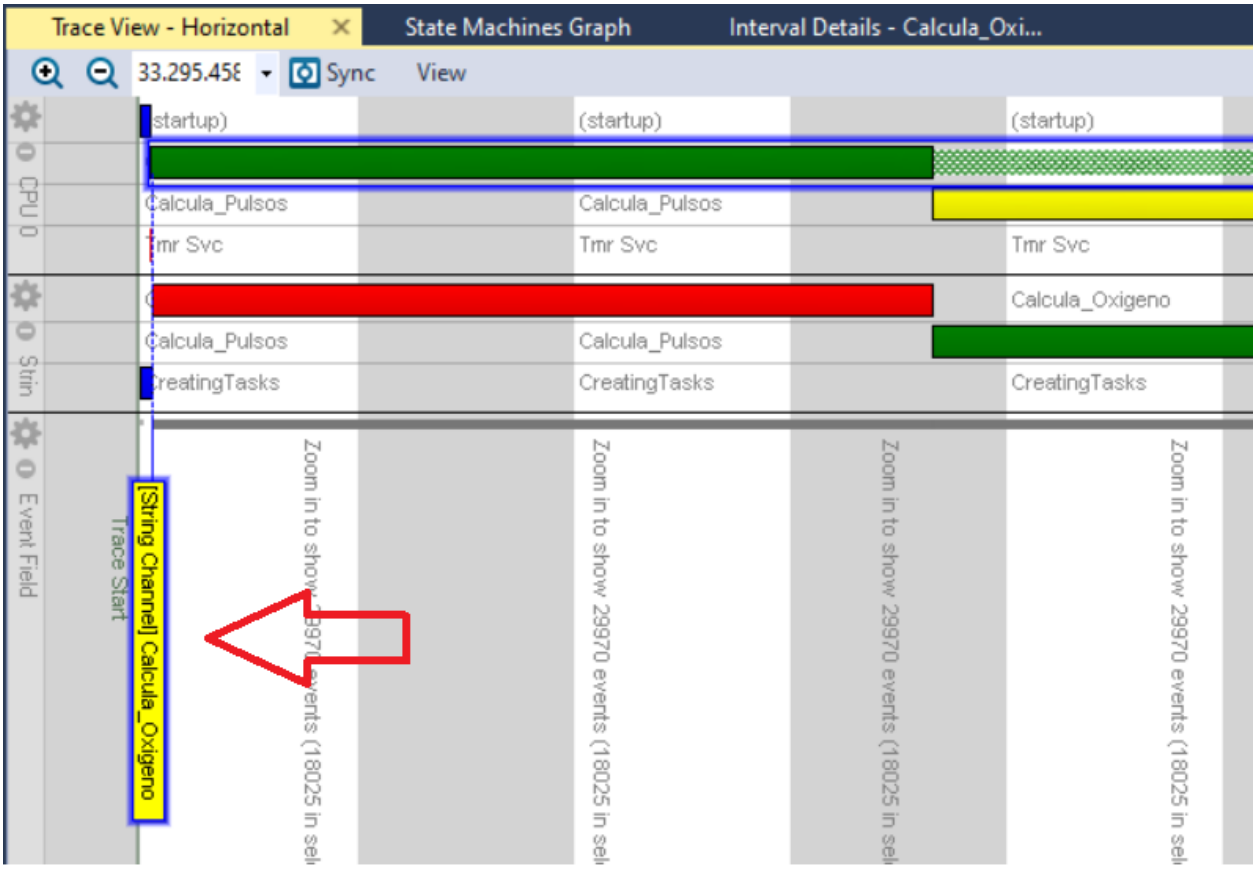



Figura 74. visualización de la transición desde *Creating Tasks* hacia *Calcula_Oxígeno*

# 9. Conclusiones

Para poder realizar la depuración con ***Tracealyzer***, en primer lugar, se ha estudiado de manera general el funcionamiento de un sistema operativo en tiempo real sobre microcontroladores, llegando a la conclusión que ***FreeRTOS*** es el más indicado porque es ligero, una configuración mínima ocupa un tamaño entorno a los 10 KB, porque es software libre y porque es compatible con el entorno de desarrollo *KEIL uVision5*.

Posteriormente se ha trabajado con el entrono *KEIL uVision5* para configurarlo de manera que pueda trabajar con ***Tracealyzer***. Durante esta etapa se descubre que los *IDEs* tienen sus limitaciones a la hora de trabajar con programas de depuración como ***Tracealyzer*** y por tanto tienen mucho por desarrollar para seguir la línea de analizar sistemas con depuradores y programas especializados en ese campo. También es cierto que el propio *IDE* posee herramientas para la depuración, pero suelen estar obsoletas o un poco limitadas.

En cuanto a ***Tracealyzer***, aunque posee mucha información y soporte por parte de su equipo técnico, sigue siendo una herramienta nueva y en continuo desarrollo, por lo que tiene muchas modificaciones o actualizaciones y está en constante cambio. Pese a las razones mencionadas anteriormente, ***Tracealyzer*** está nominado al producto del año 2021 por el magazine alemán *Electronik* en la categoría*Software Engineering* por la versión ***Tracealyzer 4.4*** *con soporte para embebido Linux*.

Respecto al proyecto sometido a depuración, en la parte *hardware* se ha destacado la importancia de tener un puerto dedicado a la programación y depuración compatible con *ULINKpro,* de lo contrario esto sería imposible. Por el lado del software, se ha aprendido a trabajar con la API de ***FreeRTOS*** y con las librerías de ***Tracealyzer*** muy fácilmente porque son muy intuitivas y escritas en lenguaje C.

Otro aspecto destacable es el de las licencias, es importante conocer las limitaciones de las mismas a la hora trabajar con proyectos que se realicen copias, modificaciones o publicaciones como se ha hecho en este proyecto final de grado.

Por último, el pequeño proyecto desarrollado, durante la etapa de desarrollo daba muchos errores y no calculaba correctamente, hasta que se leyó detenidamente la hoja de características donde se especifica claramente que el dedo debe estar a una distancia de entre 3 y 5 mm con el sensor para un funcionamiento óptimo. Para conseguirlo se ha modificado el diseño original de la carcasa añadiendo una placa que crea la distancia necesaria entre el sensor y el dedo. Para evitar ruido se ha elegido el color negro para la carcasa con el fin de evitar que la luz se refleje en alguna zona.

# 10. Pliego de Condiciones

En este apartado se hace referencia a los elementos necesarios para poder depurar con ***Tracealyzer*** un sistema embebido basado en ***FreeRTOS*** sobre plataformas Cortex-M3.

## 10.1. Elementos Hardware

Los elementos hardware hacen referencia a los componentes físicos que se han utilizado y son los siguientes:

- Tarjeta de desarrollo LPC1768-Mini-DK2
- Módulo de medición de oxígeno y pulsos por minuto MH-ET LIVE MAX30102
- Cables dupont para interconexión entre sensor y tarjeta de desarrollo.
- Cable USB 2.0 tipo-B
- Cable USB 2.0 mini-B 5 pin.
- Unidad de programación y depuración *ULINKpro* de ARM.
- PC Intel i5 de 64 bits con 8 GB de memoria RAM y al menos 5GB de espacio utilizado en el disco duro.

## 10.2. Elementos Software

Los elementos software hacen referencia a los programas y entornos de diseño, desarrollo y pruebas que se han utilizado:

- KEIL uVision5 versión 5.32.0.0
- Percepio Tracealyzer versión 4.4.1.12962
- Kernel y librerías de FreeRTOS en la versión V10.2.0
- Trace Recorder Library for Tracealyzer v4.4.1
- Microsoft Office 365 MSO (16.0.13127.21210) 64 bits
- Microsoft Windows 10 Home edition

# 11. Presupuesto

En este apartado se detalla el presupuesto necesario para llevar a cabo la depuración sobre el sistema embebido oxímetro de pulso, detallando el coste de equipamiento, el coste de mano de obra y el presupuesto total.

## 11.1. Coste por material

En esta sección se detallan los costes de las herramientas que se han utilizado. Se pueden separar en los equipos, componentes electrónicos y software.

Tabla 9. Detalle del coste de equipo y software

| Concepto | Precio | Periodo de amortización (meses) | Periodo de uso (meses) | Coste para el proyecto |
|---|---|---|---|---|
| Programa Keil uVision 5 | € - | N/A | 6 | € - |
| Licencia MDK Plus Edition | € 2.220,00 | 12 | 6 | € 1.110,00 |
| Tracealyzer Percepio 4 | € - | N/A | 6 | € - |
| Licencia Educativa Tracealyzer | € - | N/A | 6 | € - |
| PC HP Pavilion 14-ce2 | € 699,00 | 12 | 6 | € 349,50 |
| Microsoft Windows 10 | € 35,00 | 12 | 6 | € 17,50 |
| Microsoft Office 365 | € 69,00 | 12 | 6 | € 34,50 |
| **Coste total de equipo y software** | | | | € 1.511,50 |

A continuación, se muestran los costes de los componentes electrónicos.

Tabla 10. Detalle del coste del material

| Concepto | Precio/unidad | Cantidad | Coste para el proyecto |
|---|---|---|---|
| Tarjeta LPC1768-Mini-DK2 | € 21,86 | 1 | € 21,86 |
| Unidad ULINpro | € 1.111,00 | 1 | € 1.111,00 |
| Módulo MAX30102 | € 8,41 | 1 | € 8,41 |
| Tira de cables tipo dupont | € 1,97 | 1 | € 1,97 |
| Cable USB tipo A | € 2,14 | 1 | € 2,14 |
| Cable USB tipo mini-B | € 4,10 | 1 | € 4,10 |
| Impresión pieza 3D | € 15,20 | 1 | € 15,20 |
| **Coste total de equipo y software** | | | € 1.164,68 |

## 11.2. Coste por mano de obra

En esta sección se incluyen los costes de mano de obra para la parte de depuración del sistema y para la realización de la documentación.

Tabla 11. Detalle del coste mano de obra

| Concepto | Horas | €/Hora | Coste para el proyecto |
|---|---|---|---|
| Investigación acerca de FreeRTOS | 22 | 60 | € 1.320,00 |
| Estudio de la tarjeta LPC1768-Mini-DK2 | 20 | 60 | € 1.200,00 |
| Estudio herramienta Tracealyzer | 26 | 60 | € 1.560,00 |
| Estudio de Keil uVision 5 | 19 | 60 | € 1.140,00 |
| Reprografía | 1 | 27 | € 27,00 |
| Encuadernación | 1 | 15 | € 15,00 |
| **Coste total de equipo y software** | | | € 5.262,00 |

## 11.3. Presupuesto total

Teniendo en cuenta los costes reflejados en las tres tablas anteriores se obtiene el coste total de ejecución de material del proyecto, que se puede observar en la siguiente tabla.

Tabla 12. Detalle del coste de ejecución del material

| Concepto | Precio total |
|---|---|
| Coste total equipo y software | € 1.511,50 |
| Coste total del material | € 1.164,68 |
| coste total de mano de obra | € 5.262,00 |
| coste total de ejecución de material | € 7.938,18 |

A partir del coste total de ejecución de material se genera un presupuesto de contratación reflejado en la siguiente tabla.

Tabla 13. Detalle del presupuesto de contratación

| Concepto | Precio total |
|---|---|
| Coste total de ejecución de material | € 7.938,18 |
| Gastos generales | € 793,82 |
| Margen de beneficio | € 793,82 |
| Presupuesto de contratación | € 9.525,82 |

A continuación, se genera el presupuesto total del proyecto teniendo en cuenta el presupuesto de contratación y el impuesto de valor añadido.

Tabla 14. Detalle del presupuesto del proyecto

| Concepto | Precio total |
|---|---|
| Presupuesto contratación | € 9.525,82 |
| I.V.A. (21%) | € 2.000,42 |
| Presupuesto total del proyecto | € 11.526,24 |

# 12. Bibliografía


[1] Percepio Tracealyzer **https://percepio.com/tracealyzer/**

[2] Tarjeta de desarrollo, **LPC1768-Mini-DK2**

[3] Maxim Integrated Products, Inc., **High-Sensitivity Pulse Oximeter and H-R Sensor for Wearable Health**, October 2018.

[4] HAOYU Electronics, **LPC1768-Mini-DK2 Schematic**

[5] NXP B.V., **UM10360 LPC176x/5x User Manual**, December 2016.

[6] Keil uVision 5 **http://www2.keil.com/mdk5/uvision/**

[7] Sistema operativo de tiempo real ***FreeRTOS*** para microcontroladores Cortex-M **https://freertos.org/**

[8] FastBit Embedded Frain Academy, **Mastering RTOS: Hands on FreeRTOS and STM32Fx with debugging**, Udemy

[9] Israel Gbati, **FreeRTOS from Ground Up on ARM Processors**, Udemy

[10] Richard Barry, **Mastering the *FreeRTOS* real time kernel** , A hands-On Tutorial Guide , 2016.

[11] Richard Barry, **Using The FreeRTOS Real Time Kernel, LPC17xx Edition, 2010.**

[12] Amazon.com, Inc. or its affiliates, **The FreeRTOS Reference Manual**, version 10.0.0 , 2017.

# 13. Links a recursos gráficos empleados en el TFG


[1] https://tiendaelite.com/wp-content/uploads/2020/04/pulsometro-principal.jpg

[2] https://www.hotmcu.com/lpc1768minidk2-development-board-28-tft-lcd-p-12.html

[3] https://es.aliexpress.com/item/32946413462.html

[4] https://datasheets.maximintegrated.com/en/ds/MAX30102.pdf

[5] https://datasheets.maximintegrated.com/en/ds/MAX30102.pdf

[6] https://datasheets.maximintegrated.com/en/ds/MAX30102.pdf

[7] https://datasheets.maximintegrated.com/en/ds/MAX30102.pdf

[8] https://datasheets.maximintegrated.com/en/ds/MAX30102.pdf

[9] https://www.haoyuelectronics.com/Attachment/LPC1768-Mini-DK2/LPC1768-Mini-DK2_Schematic.pdf

[10] https://www.nxp.com/docs/en/user-guide/UM10360.pdf

[11] https://www.keil.com/products/ulinkpro/default.asp

[12] https://freertos.org/

[13] https://docs.aws.amazon.com/freertos/latest/userguide/what-is-freertos.html

[14] https://www.dsi.fceia.unr.edu.ar/images/Presentacion_FreeRTOS.pdf

[15] https://datasheets.maximintegrated.com/en/ds/MAX30102.pdf

[16] https://datasheets.maximintegrated.com/en/ds/MAX30102.pdf

[17] https://percepio.com/docs/FreeRTOS/manual/

[18] https://percepio.com/docs/FreeRTOS/manual/

# 14. Anexos

## 14.1. Código del fichero “main.c”

```
/*****************************************************************************************************
                        FREERTOS FOR ARM uCONTROLLERS M3-CORTEX
*****************************************************************************************************
 * File:              main.c (ARM controllers FreeRTOS)
 * Version:    FreeRTOS Kernel V10.2.0
 * Author:            Nelson Ismael Rivero Meneses
 * Website:
 * Description: Program to demonstrate the task switching between the three tasks to understand
                            working of scheduler.

GNU GENERAL PUBLIC LICENSE:
    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program.  If not, see <http://www.gnu.org/licenses/>.

 *****************************************************************************************************/

/* Kernel includes. */
#include "FreeRtOSConfig.h"          /* Must come first. */
#include "FreeRTOS.h"
#include "timers.h"
#include "task.h"                                   /* RTOS task related API prototypes. */


/* Add any manufacture supplied header files can be included
here. */
/* #include "hardware.h" */

#include <lpc17xx.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "cmsis_os2.h"                      /*  CMSIS RTOS header file. */

#include "main.h"



#define        RATE_SIZE      10                            //Increase this for more averaging. 4 is
good.
uint8_t        rates[RATE SIZE];            //Array of heart rates
uint8_t        rateSpot = 0;
uint32_t       lastBeat = 0;                //Time at which the last beat occurred
float          beatsPerMinute;
int            beatAvg;

uint32_t irBuffer[100]; //infrared LED sensor data
uint32_t redBuffer[100];  //red LED sensor data


int32_t bufferLength; //data length
int32_t spo2; //SPO2 value
int8_t validSPO2; //indicator to show if the SPO2 calculation is valid
int32_t heartRate; //heart rate value
int8_t validHeartRate; //indicator to show if the heart rate calculation is valid
```

```
uint8_t ledBrightness = 60; //Options: 0=Off to 255=50mA
uint8_t sampleAverage = 4; //Options: 1, 2, 4, 8, 16, 32
uint8_t ledMode = 2; //Options: 1 = Red only, 2 = Red + IR, 3 = Red + IR + Green
uint8_t sampleRate = 100; //Options: 50, 100, 200, 400, 800, 1000, 1600, 3200
int pulseWidth = 411; //Options: 69, 118, 215, 411
int adcRange = 4096; //Options: 2048, 4096, 8192, 16384

uint32_t       delta;
long           irValue;
#define        LENGTH_STR     100
char           str[LENGTH_STR];
char           str_lcd[LENGTH_STR];
#define        true    1
#define        false   0

uint8_t        state_PPM,state_oxi,buffr[25];
volatile       mystruct Flag;

xTaskHandle TaskHandle_1;
xTaskHandle TaskHandle_2;

/* Canales ITM creados para depuración */

traceString MyChannel;
traceString Channel_SPO2;
traceString Channel_PPM;
traceString Channel_Clock;

/* Priorities at which the tasks are created.  The event semaphore task is
given the maximum priority of ( configMAX_PRIORITIES - 1 ) to ensure it runs as
soon as the semaphore is given. */

#define TASK_1_PRIORITY      ( tskIDLE_PRIORITY + 2 )
#define TASK_2_PRIORITY      ( tskIDLE_PRIORITY + 3 )


/*-----------------------------------------------------------------------------------------------------*/

/*
 * Implement this function for any hardware specific clock configuration
 * that was not already performed before main() was called.
 */
static void prvSetupHardware( void );

static void MyTask1( void* pvParameters );
static void MyTask2( void* pvParameters );


uint8_t comprueba_sensor(void);
void update_time(void);

/*-----------------------------------------------------------------------------------------------------*/

int main(void)
{
    /* Configure the system ready to run the demo.  The clock configuration
    can be done here if it was not done before main() was called. */
    prvSetupHardware();


    /* Create task1 as described in the comments at the top
    of this file. */
    xTaskCreate(     /* The function that implements the task. */
                    MyTask1,
                    /* Text name for the task, just to help debugging. */
                    ( signed char * ) "Calcula_Oxigeno",
                    /* The size (in words) of the stack that should be created
                    for the task. */
                    configMINIMAL_STACK_SIZE,
                    /* A parameter that can be passed into the task.  Not used
                    in this simple demo. */
                    NULL,
                    /* The priority to assign to the task.  tskIDLE_PRIORITY
                    (which is 0) is the lowest priority.  configMAX_PRIORITIES - 1
                    is the highest priority. */
                    TASK_1_PRIORITY,
                    /* Used to obtain a handle to the created task.  Not used in
```

```
                    this simple demo, so set to NULL. */
                    &TaskHandle_1 );


    /* Create task2 as described in the comments at the top
    of this file. */
    xTaskCreate(     MyTask2,
                    ( signed char * ) "Calcula_Pulsos",
                    configMINIMAL_STACK_SIZE,
                    NULL,
                    TASK_2_PRIORITY,
                    &TaskHandle_2 );




    /* Start the tasks and timer running. */

                vTaskSuspend(TaskHandle_2);

                vTaskStartScheduler();


    /* If all is well, the scheduler will now be running, and the following line
    will never be reached.  If the following line does execute, then there was
    insufficient FreeRTOS heap memory available for the idle and/or timer tasks
    to be created.  See the memory management section on the FreeRTOS web site
    for more details.  */
    for( ;; );

}

/*-----------------------------------------------------------------------------------------------*/

static void prvSetupHardware( void )
{
    /* Ensure all priority bits are assigned as preemption priority bits
    if using a ARM Cortex-M microcontroller. */
    NVIC_SetPriorityGrouping( 0 );

                /* Setup the clocks, etc. here, if they were not configured before main() was called.
*/


                MyChannel = xTraceRegisterString("String Channel");
                Channel_SPO2 = xTraceRegisterString("SPO2");
                Channel_PPM = xTraceRegisterString("PPM");
                Channel_Clock = xTraceRegisterString("Clock");

                vTraceEnable(TRC_START);
                vTracePrint(MyChannel, "CreatingTasks");
                vTracePrintF(Channel_SPO2, "%3d",0);
                vTracePrintF(Channel_PPM, "%3d",0);

                SystemInit();                   /* Initialize the controller */
                uart0_init(115200);
                tx_completa_0 = 1;
                uart0_menu();
                PWM_Init();
                PWM_Set(0);

                LED_PORT3;

                lcdInitDisplay();
                fillScreen(BLACK);

                if(comprueba_sensor() == 1)   /* Error sensor no detected */
                       while(1);

                update_time();
                TIMER1_Init();
                init_RTC();

                LED_Task1_Off;
                LED_Task2_Off;
```

```
}
/*-----------------------------------------------------------------------------------------------------*/

static void MyTask1( void* pvParameters )
{

	for(;;){
		vTracePrint(MyChannel, "Calcula_Oxigeno");

		/* Led to indicate the execution of Task1*/
		LED_Task1_On;

		tx_cadena_UART0("Calcula_Oxígeno \r\n");
		drawString(200, 10,"                ", WHITE, BLACK, LARGE);
		drawString(200, 10, "Calcula Ox~geno", CYAN, BLACK, LARGE);

		/* Configure sensor with these settings */
		MAX30105_setup(ledBrightness, sampleAverage, ledMode, sampleRate, pulseWidth,
adcRange);
		init_max30105();
		state oxi = state checkFinger;
		while (calcula_oxigeno() == 0) {

			if (Flag.clock) {
					Flag.clock = 0;
					actualiza_rtc();
					}

			if (Flag.NoFinger) {
					Flag.NoFinger = 0;
					escribe_nofinger();
					}
		}
		/* Print SPO2 value*/
		escribe_spo2();
		drawString(200, 10,"                ", WHITE, BLACK, LARGE);
		LED_Task1_Off;
		vTaskResume(TaskHandle_2);

	}

}

/*-----------------------------------------------------------------------------------------------------*/

static void MyTask2( void* pvParameters )
{

  for(;;){
		vTracePrint(MyChannel, "Calcula_Pulsos");
		/* Led to indicate the execution of Task2*/
		LED_Task2_On;
		Flag.EntryTask2 = 1;
		tx_cadena_UART0("Calcula_Pulsos\r\n");

		drawString(100, 10, "                ", CYAN, BLACK, LARGE);
		drawString(100, 10, "Calcula Pulsos", CYAN, BLACK, LARGE);

		MAX30105_setup(31,4, 3, 400, 411, 4096);
		setPulseAmplitudeRed(0x0A); //Turn Red LED to low to indicate sensor is running
		setPulseAmplitudeGreen(0); //Turn off Green LED


		state_PPM = state_checkFinger;
		while (calcula_PPM() == 0) {

			if (Flag.Print_PPM) {
				Flag.Print_PPM = 0;
				escribe_ppm();
				}
			if (Flag.NoFinger) {
				Flag.NoFinger = 0;
				escribe_nofinger();
				}
			if (Flag.clock) {
				Flag.clock = 0;
				actualiza_rtc();
```

```
                        }
                }

                drawString(150, 10, "                    ", CYAN, BLACK, LARGE);
                drawString(100, 10, "                    ", CYAN, BLACK, LARGE);
                drawString( 50, 10, "                    ", CYAN, BLACK, LARGE);

                LED_Task2_Off;


                vTaskSuspend(TaskHandle_2);
        }

}

/*-------------------------------------------------------------------------------------------------------*/

void update_time(void)
{
        sprintf(buffr,"%02d/%02d/%04d   %02d:%02d",
        LPC_RTC->DOM,LPC_RTC->MONTH,LPC_RTC->YEAR,LPC_RTC->HOUR,LPC_RTC->MIN);
        drawString(10, 10, buffr, WHITE, BLACK, LARGE);
}

/*-------------------------------------------------------------------------------------------------------*/

uint8_t comprueba_sensor(void)
{
                if (readPartID() != 0x15)
                {
                drawString(200, 10, "No sensor. Reboot.", CYAN, BLACK, LARGE);
                tx_cadena_UART0("No sensor. Reboot.\r\n");
                return 1;
                }
                return 0;

}


void actualiza_rtc(void){
                        sprintf(buffr,"%02d/%02d/%04d   %02d:%02d",
                        LPC_RTC->DOM,LPC_RTC->MONTH,LPC_RTC->YEAR,LPC_RTC->HOUR,LPC_RTC->MIN);

                        vTracePrintF(Channel_Clock, buffr, beatAvg);
                        update_time();
}


void escribe_spo2(void){

                        vTracePrintF(Channel_SPO2, "%3d",spo2);
                        sprintf((char *)str,"\r\n SpO2:%3d %% \r\n\r\n",spo2);
                        memset(str_lcd, 0, 100);
                        sprintf((char *)str_lcd,"SpO2:%3d %% ",spo2);
                        tx_cadena_UART0((char *)str);
                        drawString(150, 10, "                    ", CYAN, BLACK, LARGE);

                        drawString(150, 10, str_lcd, CYAN, BLACK, LARGE);
                        Flag.Print_SPO2 = 0;
                        PWM_Set(50);
                        delay_lpc(200);
                        PWM_Set(0);
}


void escribe_ppm(void){

                        static int _a=0;
                        if (Flag.EntryTask2 == 1) {
                                Flag.EntryTask2 = 0;
                                _a = 0;
                                return ;
                        }

                        sprintf((char *)str,"\r\n PPM: %d \r\n", beatAvg);
                        sprintf((char *)str_lcd,"PPM: %d ", beatAvg);
                        tx_cadena_UART0((char *)str);
```

```
                        drawString( 50, 10,  "                  ", CYAN, BLACK, LARGE);

                        drawString( 50, 10, str_lcd, CYAN, BLACK, LARGE);
                        vTracePrintF(Channel_PPM, "%3d",beatAvg);

                        escribe_puntos(_a++);
                        drawString(100, 235, str_lcd, CYAN, BLACK, LARGE);

                        PWM_Set(50);
                        delay_lpc(200);
                        PWM_Set(0);
}


void escribe_nofinger(void){

                        tx_cadena_UART0("Por favor introduzca su dedo\r\n");
                        drawString( 50, 10, "                   ", CYAN, BLACK, LARGE);
                        drawString( 50, 10, "Introduzca su dedo", RED, WHITE, LARGE);


/* END OF FILE */
```

Universidad de Alcalá

Escuela Politécnica Superior



ESCUELA POLITECNICA SUPERIOR